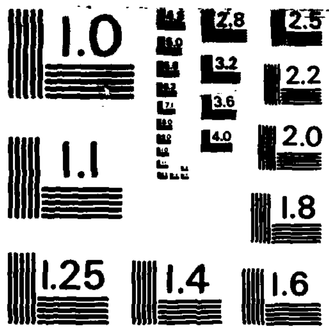
AD-A139 918 AUTOMATING THE TRANSFORMATIONAL DEVELOPMENT OF SOFTWARE 1/3
VOLUME 2 APPENDICES(U) UNIVERSITY OF SOUTHERN
CALIFORNIA MARINA DEL REY INFORMATION S.. S F FICKAS
UNCLASSIFIED MAR 83 ISI/RR-83-109 NSF-MCS79-18792 F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD A139918

(5)

ISI/RR-83-109
March 1983

Stephen F. Fickas

University of Southern California

Automating the Transformational Development of Software (Appendices) Volume 2

INFORMATION SCIENCES INSTITUTE

213/822-1511
4676 Admiralty Way/Marina del Rey/California 90291-6695

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ISI/RR-83-109	2. GOVT ACCESSION NO. AD-A139918	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Automating the Transformational Development of Software (Appendices) Volume 2		5. TYPE OF REPORT & PERIOD COVERED Research Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Stephen F. Fickas		8. CONTRACT OR GRANT NUMBER(s) MCS-7918792
9. PERFORMING ORGANIZATION NAME AND ADDRESS USC/Information Sciences Institute 4676 Admiralty Way Marina del Rey, CA 90291		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS National Science Foundation 1800 G St. N.W. Washington, D.C. 20550		12. REPORT DATE March 1983
		13. NUMBER OF PAGES 280
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

This document is approved for public release and sale; distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

....

18. SUPPLEMENTARY NOTES

This report was the author's Ph.D. dissertation at the University of California, Irvine, Department of Information and Computer Science. The author's current address is Department of Computer Science, University of Oregon, Eugene, OR 97403.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

automated software development, automation and documentation of software development, interactive software development system, problem solving, transformational implementation

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

(OVER)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT

This report proposes a new model of software development by transformation. It provides a formal basis for automating and documenting the software development process. The current manual transformation model has two major problems: 1) long sequences of low-level transformations are required to move from formal specification to implementation, and 2) the problem-solving used to reach an implementation is not recorded. Left implicit (and undocumented) are the goals and methods that lead to transformation applications, and the criteria used to select one transformation over another. The new model, as incorporated in a system called Glitter, explicitly represents transformation goals, methods, and selection criteria. Glitter achieves a user-supplied goal by carrying out the problem-solving required to generate an appropriate sequence of transformation applications. For example, the user asks Glitter to eliminate a data structure that would be expensive to store or a function costly to compute. Glitter achieves this by locating all references to the offending construct and devising an appropriate substitution for each. Glitter was able to automatically generate 90 percent of the planning and transformation steps in the examples studied. This report is published in two volumes. Volume 1 contains the text of the report; Volume 2 is a set of seven appendices relating to and illustrating the text in Volume 1.

ISI/RR-83-109
March 1983

University of Southern California

Stephen F. Fickas

Automating the Transformational Development of Software (Appendices) Volume 2

INFORMATION SCIENCES INSTITUTE
213/822-1511
4676 Admiralty Way/Marina del Rey/California 90291-6695

This research is supported by the National Science Foundation under Contract No. MCS-7918792. Views and conclusions contained in this report are the author's and should not be interpreted as representing the official opinion or policy of NSF, the U.S. Government, or any person or agency connected with them.

Contents

Appendix A
Gist specification of package router

In this appendix, we present the formal Gist specification of the package router problem. The English description is given in section 3.1, page 38. An overview of the specification is given in Chapter 4. The original router specification is due to Feather and London [London & Feather 82]; the version here incorporates some minor improvements.

Key to font conventions and special symbols used in Gist

symbol	meaning	example
|	of type	*obj* | T · object *obj* of type T
||	such that	(an *integer* || (*integer* > 3)) · an integer greater than 3
_	may be used to build names, like this_name	
.	concatenates a type name with a suffix to form a variable name, e.g. *integer.1* Variables with distinct suffices denote distinct objects.	

fonts	meaning	example
underlined	key word	begin, definition, if
SMALL CAPITALS	type name	INTEGER
lower case italics	variable	*x*
UPPER CASE BOLDFACE	action, demon, relation and constraint names	**SET_SWITCH**
Mixed Case Boldface	attribute names	**Destination**

Package Router Specification in Gist

The network hardware

type LOCATION() supertype of

< SOURCE(**source_outlet** | PIPE);

Gist comment - the above line defines SOURCE to be a type with one attribute, **source_outlet**, and only objects of type PIPE may serve as such attributes. end comment

```
  PIPE(connection_to_switch_or_bin | (SWITCH union BIN) );

  SWITCH(switch_outlet | PIPE :2, switch_setting | PIPE)
         where always required
               switch:switch_setting = switch:switch_outlet end;
  BIN()
>;
```

Spec comment - of the above types and attribute, only the SWITCH_SETTING attribute of SWITCH is dynamic in this specification, the others remain fixed throughout. end comment

Gist comment - by default, attributes (e.g. SOURCE_OUTLET) of types (e.g. SOURCE) are functional - (e.g. there is one and only one pipe serving as the SWITCH_SETTING attribute of the SOURCE). The default may be overridden, as occurs in the SWITCH_OUTLET attribute of SWITCH - there the ":2" indicates that each switch has exactly 2 pipes serving as its SWITCH_OUTLET attribute. end comment

```
always prohibited MORE_THAN_ONE_SOURCE
  exists source.1, source.2;
```

Gist comment - constraints may be stated as predicates following either always required (in which case the predicate must always evaluate to true), or always prohibited (in which case the predicate must never evaluate to true). The usual logical connectives, quantification, etc. may be used in Gist predicates. Distinct suffixes on type names after exists have the special meaning of denoting distinct objects. end comment

```
always required PIPE_EMERGES_FROM_UNIQUE_SWITCH_OR_SOURCE
  for all pipe ||
    ( exists unique switch_or_source | (SWITCH union SOURCE) ||
      (pipe = switch_or_source:switch_outlet or
       pipe = switch_or_source:source_outlet ) );
```

Gist comment - the values of attributes can be retrieved in the following manner: if obj is an object of type T, where type T has an attribute ATT, then obj:ATT denotes any object serving as obj's ATT attribute. end comment

```
always required UNIQUE_PIPE_LEADS_INTO_SWITCH_OR_BIN
  for all switch_or_bin | (SWITCH union BIN) ||
    ( exists unique pipe ||
       ( pipe:connection_to_switch_or_bin = switch_or_bin ) );
```

```
relation LOCATION_ON_ROUTE_TO_BIN(LOCATION,BIN)
  definition
    case LOCATION of
      BIN => LOCATION = BIN;
      PIPE => LOCATION_ON_ROUTE_TO_BIN(LOCATION:connection_to_switch_or_bin,BIN);
      SWITCH => LOCATION_ON_ROUTE_TO_BIN(LOCATION:switch_outlet,BIN);
      SOURCE => LOCATION_ON_ROUTE_TO_BIN(LOCATION:source_outlet,BIN);
    end case;
```

Development comment - mapped at step 5.4 end comment

Spec comment - this relation is defined to hold between a location and bin if and only if the location lies on route to the bin, i.e. the location is the bin, or the location is a pipe connected to a location leading to the bin (a recursive definition), or a switch either of the outlets of which leads to the bin, or a source whose outlet leads to the bin. end comment

Gist comment - the predicate of a defined relation denotes those tuples of objects participating in that relation. For any tuple of objects of the appropriate types, that tuple (in the above relation, a 2-tuple of *LOCATION* and *BIN*) is in the defined relation if and only if the defining predicate equals true for those objects. end comment

```
always required SOURCE_ON_ROUTE_TO_ALL_BINS
  for all bin || LOCATION_ON_ROUTE_TO_BIN(the source,bin) ;
```

Packages - the objects moving through the network

```
type PACKAGE(located_at | LOCATION, destination | BIN) ;

relation MISROUTED(PACKAGE)
  definition
    ~ LOCATION_ON_ROUTE_TO_BIN(PACKAGE:located_at, PACKAGE:destination) or
    SWITCH_SET_WRONG_FOR_PACKAGE(PACKAGE:located_at,PACKAGE) ;
```

Development comment - mapped at step 5.5 end comment

Spec comment - a package is misrouted if it is at a location not on route to its destination, or in a switch set the wrong way. end comment

Implementable Portion

<u>Spec</u> <u>comment</u> - the portion over which we have control, and are to implement. <u>end</u> <u>comment</u>

<u>agent</u> PACKAGE_ROUTER() <u>where</u>

<u>relation</u> **PACKAGES_EVER_AT_SOURCE**(*PACKAGE_SEQ* | <u>sequence</u> <u>of</u> PACKAGE)
  <u>definition</u> *PACKAGE_SEQ* =
    ({*package* || (*package*:**located_at** = <u>the</u> *source*) <u>asof</u> <u>ever</u>}
      <u>ordered</u> <u>temporally</u> <u>by</u> <u>start</u> (*package*:**located_at** = <u>the</u> *source*));

<u>Development</u> <u>comment</u> - mapped at step 1.10 <u>end</u> <u>comment</u>

<u>Spec</u> <u>comment</u> - the sequence of packages ever to have been located at the source, in the order in which they were there. <u>end</u> <u>comment</u>

The source station

```
demon RELEASE_PACKAGE_INTO_NETWORK(package.new)
  trigger package.new:located_at = the source
  response
    begin
     if (the package.previous || ( package.previous immediately < package.new
                                    wrt PACKAGES_EVER_AT_SOURCE(*) )
          ):destination ≠ package.new:destination
     then WAIT[] ;
```

Development comment - part of final implementation end comment

Spec comment - must delay release of the new package unless the immediately preceding package was destined for the same bin. end comment

```
     update :located_at of package.new to (the source):source_outlet
    end ;
```

Gist comment - a demon is a data-triggered process. Whenever a state change takes place in which the value of demon's trigger predicate changes from false to true, the demon is triggered, and performs its response.

The use of a relation with a '*' filling one of its positions denotes any object that could fill that position. Thus R(..,*,..) for relation R is equivalent to an *obj* || R(..,*obj*,..) end comment

The switches

```
relation SWITCH_IS_EMPTY(switch)
  definition ~ exists package || package:located_at = switch;
```

Development comment - unfolded at step 6.10 end comment

```
demon SET_SWITCH(switch)
  trigger RANDOM()
  response
    begin
      require SWITCH_IS_EMPTY(switch);
      update :switch_setting of switch to switch:switch_outlet
    end;
```

Development comment - mapped at step 6.1 end comment

Spec comment - the non-determinism of when and which way to set switches is constrained by the always prohibited that follows shortly: end comment

```
relation PACKAGES_DUE_AT_SWITCH(PACKAGES_DUE | sequence of PACKAGE, SWITCH)
  definition
    PACKAGES_DUE =
      { a package ||
        LOCATION_ON_ROUTE_TO_BIN(SWITCH,package:destination) and
        ~ ((package:located_at = SWITCH) asof ever) and
        ~ MISROUTED(package)
      } ordered wrt start (package:located_at = the source)
```

Development comment - mapped at step 5.1 end comment

Spec comment - packages due at a switch are those packages for whom (i) the switch lies on their route to their destinations, (ii) they have not already reached the switch, and (iii) they are not misrouted. They are ordered by the order in which they were at the source. end comment

```
relation SWITCH_SET_WRONG_FOR_PACKAGE(SWITCH, PACKAGE)
  definition
   LOCATION_ON_ROUTE_TO_BIN(SWITCH,PACKAGE:destination) and
   ~ LOCATION_ON_ROUTE_TO_BIN(SWITCH:switch_setting,PACKAGE:destination) ;
```

Development comment - mapped at step 5.8 end comment

Spec comment - A switch is set wrong for a package if the switch lies on the route to that package's destination, but the switch is set the wrong way. end comment

```
always prohibited DID_NOT_SET_SWITCH_WHEN_HAD_CHANCE
  exists package, switch ||
    (package:located_at = switch
     and
     SWITCH_SET_WRONG_FOR_PACKAGE(switch,package)
     and
     ((package = first(PACKAGES_DUE_AT_SWITCH(*,switch)) and
       SWITCH_IS_EMPTY(switch)) asof ever)
    );
```

Development comment - mapped at step 4.1 end comment

Spec comment - must never reach a state in which a package is in a wrongly set switch, if there has been an opportunity to set the switch correctly for that package, i.e. at some time that package was the first of those due at the switch and the switch was empty. end comment

Arrival of misrouted package

```
demon MISROUTED_PACKAGE_REACHED_BIN(package,bin.reached,bin.intended)
  trigger package:located_at = bin.reached and package:destination = bin.intended
  response  MISROUTED_ARRIVAL[ bin.reached, bin.intended ];
```

Development comment - mapped at step 6.13 end comment

```
action MISROUTED_ARRIVAL[ bin.reached, bin.intended ]
```

Development comment - part of implementation end comment

The environment

agent ENVIRONMENT() where

Arrival of packages at source

demon CREATE_PACKAGE()
trigger RANDOM()
response
create *package.new* || (*package.new*:**destination** = a *bin* and
package.new:**located_at** = the *source*) ;

Spec comment - for the purposes of defining the environment in which the package router is to operate, packages arrive at random intervals at the source with random destinations, subject to the following constraint. end comment

always prohibited MULTIPLE_PACKAGES_AT_SOURCE
exists *package.1*, *package.2* ||
package.1:**located_at** = the *source* and *package.2*:**located_at** = the *source* ;

Movement of packages through network

relation MOVEMENT_CONNECTION(*LOCATION.1*, *LOCATION.2*)
definition
(case *LOCATION.1* of
PIPE => *LOCATION.1*:**connection_to_switch_or_bin**;
SWITCH => *LOCATION.1*:**switch_setting**
end case) = *LOCATION.2*;

demon MOVE_PACKAGE(*package*)
trigger ∃ *location.next* || MOVEMENT_CONNECTION(*pacakge*:LOCATED_AT, *location.next*)
response
update :**located_at** of *package* to MOVEMENT_CONNECTION(*package*:**located_at**, *);

Spec comment - this demon models the unpredictable movement of packages through the network.It triggers when a package has some place to move to (all cases except when in a bin) and at some arbitrary time in the future moves it there. end comment

```
always prohibited PACKAGES_OVERTAKING_ONE_ANOTHER
  exists package.1, package.2, location
   || start ( package.1:located_at = location ) earlier than
        start ( package.2:located_at = location ) and

      finish ( package.2:located_at = location ) earlier than
        finish ( package.1:located_at = location ) ;
```

Spec comment - we are assured that packages do not overtake one another while they are moved through the network: a package which enters a location (switch, pipe, source) eralier than another does not exit later. end comment

```
action WAIT[] ;
```

Observable environment

Spec comment - portions of environment to be used to describe observable information available to implementor. end comment

```
type SENSOR() supertype of < switch(); bin() > ;

demon PACKAGE_ENTERING_SENSOR(package,sensor)
  trigger package:located_at = sensor
  response null ;

demon PACKAGE_LEAVING_SENSOR(package,sensor)
  trigger ~ package:located_at = sensor
  response null

end
```

Implementation Specification

__Spec__ __comment__ - this section is intended to capture the requirements placed on an implementor of the package router agent. __end__ __comment__

```
implement PACKAGE_ROUTER
  observing
    attributes
      source_outlet,
      connection_to_switch_or_bin,
      switch_outlet,
      package:destination when package:located_at = the source,
      package:located_at when package:located_at = the source ;

    events
      PACKAGE_ENTERING_SENSOR($,sensor),
      PACKAGE_LEAVING_SENSOR($,sensor) ;

  effecting
    attributes
      switch_setting,
      package:located_at when package:located_at = the source ;


  exporting
    events
      MISROUTED_ARRIVAL(bin.reached,bin.intended)
      WAIT[];

end implement:
```

Appendix B
Development Goal-Structure

In this appendix, we explicate the implicit goal structure of the router development of appendix C and further, provide a broad outline of that development. The sectioning of the appendix follows that of appendix C. Each step takes the following form:

Level StepNum *Goal* <arguments>
Method

The level, a positivie interger, represents the goal nesting level. This is also provided visually by indentation. Goals at level 0, i.e. goals posted by the user, have no level printed. All goals posted by the user are underlined. A goal's <arguments> are generally printed in abbreviated form so as to fit on a single line. The method printed below the goal is the one chosen in the development.

B.1. Remove PACKAGES_EVER_AT_SOURCE

1.1 *Remove* peas from spec

RemoveRelation

1 1.2 *Remove* reference to packages_ever_at_source (peas) from spec

MegaMove

2 1.3 *Isolate* derived object

FoldGenericIntoRelation

3 1.4 *Globalize* derived object

GlobalizeDerivedObject

4 1.5 (try) *Reformulate* p.new as global

ReformulateLocalAsLast

5 1.6 *Reformulate* p.new as last(peas(*))

Ø

6 1.7 *Manual* manual-replace(p.new last(peas))

manual step

2 1.8 *MaintainIncrementally* previous_package

ScatterMaintenanceForDerivedRelation

3 1.9 *Flatten* previous_package

Flatten

4 1.10 *Map* peas

MaintainDerivedRelation

5 1.11 *MaintainIncrementally* peas

IntroduceSeqMaintenanceDemon

1 1.12 *Remove* reference peas from spec

PositionalMegaMove

2 1.13 *Reformulate* derived-object as positional retrieval

ReformulateDerivedObject

3 1.14 *Reformulate* relative retrieval as equivalence relation

ReformulateRelativeRetrievalAsLast

4 1.15 *Equivalence* last(peas@p) and p

Anchor2

5 1.16 *Reformulate* last(peas@p) as p

ReformulateAsObject

2 1.17 *Isolate* last(peas)

FoldGenericIntoRelation

2 1.18 *MaintainIncrementally* last_package

ScatterMaintenanceForDerivedRelation

1 1.19 *Remove* reference peas from spec

RemoveByObjectizingContext

2 1.20 *Reformulate* last(peas@p) as object

ReformulateAsObject

1 1.21 *Remove* update peas from spec

RemoveUnusedAction

2 1.22 *Show* update unnoticed

ShowDysteleological

B.2. Remove PREVIOUS_PACKAGE

2.1 *Remove* previous_package

RemoveRelation

1 2.2 *Remove* reference previous_package from spec

ReplaceRefWithValue

2 2.3 *Show* value known of previous_package

ShowUpdateGivesValue

2 2.4 *Show* last_package still holds at conditional

ShowNewValueStillValid

3 2.5 *Show* last_package doesn't change

MoveInterveningUpdate

4 2.6 *ComputeSequentially* update of last_package after conditional

MoveOutOfAtomic

5 2.7 *Unfold* atomic

UnfoldAtomic

5 2.8 (reposted) *ComputeSequentially* update of last_package after conditional

ConsolidateToMakeSequential

6 2.9 *Consolidate* notice_new_package_at_source and release_package_into_network

MergeDemons

7 2.10 *Equivalence* declaration lists

EquivalenceCompoundStructures

8 2.11 *Equivalence* p and p.new

Anchor2

9 2.12 *Reformulate* p as p.new

RenameVar

5 2.13 (reposted) *ComputeSequentially* update of last_package after conditional

SwapUp

6 2.14 *Swap* update of last_package with conditional

SwapStatements

B.3. Remove LAST_PACKAGE

3.1 *Remove* last_package

RemoveRelation

1 3.2 *Remove* reference last_package from spec

MegaMove

2 3.3 *Isolate* last_package:destination

FoldGenericIntoRelation

2 3.4 *MaintainIncrementally* last_package_destination

ScatterMaintenanceForDerivedRelation

1 3.5 *Remove* update of last_package

RemoveUnusedAction

B.4. Map DID_NOT_SET_SWITCH_WHEN_HAD_CHANCE

4.1 <u>*Map*</u> did_not_set_switch_when_had_chance

MapConstraintAsDemon

1 4.2 *Show* body implies Q

ConjunctImpliesConjunctArm

1 4.3 *Map* set_switch_when_have_chance (sswhc)

MapByConsolidation

2 4.4 *Consolidate* sswhc and set_switch

MergeDemons

3 4.5 *Equivalence* two triggers

Anchor2

4 4.6 *Reformulate* random as specific

SpecializeRandom

4.7 <u>*Map*</u> require ~P from ThisEvent until EverMore

CasifyPosConstraint

1 4.8 *Casify* require ~P from ThisEvent until EverMore

CasifyFromUntilEverConstraint

1 4.9 *Map* require ~P at ThisEvent

TriggerImpliesConstraint

1 4.10 *Map* require ~P after ThisEvent

CasifyPosConstraint

2 4.11 *Casify* require ~P after ThisEvent

CasifyAroundEvent

2 4.12 *Map* require ~P after ThisEvent until E

NotXUntilX

2 4.13 *Map* ~P during E

CasifyPosConstraint

3 4.14 *Casify* require ~P during E

PastInduction

3 4.15 *Map* require ~P at last update switch_setting

MoveConstraintToAction

3 4.16 *Map* require ~(start ~P) between last update, E

ShowNoChange

4 4.17 *Show* ~(start ~P) between last update, E

Ø

4.18 *Map* update of switch_setting where P

ComputeNewValue

4.19 *Unfold* switch_set_wrong_for_package at set_switch

ComputeNewValue

B.5. Map PACKAGES_DUE_AT_SWITCH

5.1 *Map* packages_due_at_switch (pdas)

MaintainDerivedRelation

1 5.2 *MaintainIncrementally* pdas

ScatterMaintenanceForDerivedRelation

2 5.3 *Flatten* pdas

Flatten

3 5.4 *Map* location_on_route_to_bin

StoreExplicitly

3 5.5 *Map* misrouted

UnfoldDerivedRelation

4 5.6 *Unfold* misrouted at pdas

ScatterComputationOfDerivedRelation

2 5.7 *Flatten* pdas

Flatten

3 5.8 *Map* switch_set_wrong_for_package

UnfoldDerivedRelation

4 5.9 *Unfold* switch_set_wrong_for_package

ScatterComputationOfDerivedRelation

1 5.10 *Purify* loop in create_package

PurifyDemon

2 5.11 *Remove* loop from create_package

RemoveFromDemon

3 5.12 *Globalize* loop in create_package

GlobalizeAction

4 5.13 *Unfold* atomic

UnfoldAtomic

1 5.14 *Purify* conditional in move_package

PurifyDemon

2 5.15 *Remove* conditional in move_package

RemoveFromDemon

3 5.16 *Globalize* conditional in move_package

GlobalizeAction

4 5.17 *Unfold* atomic

UnfoldAtomic

5.18 *Casify* package_leaving_sensor

CasifySuperTrigger

5.19 *Casify* package_entering_sensor

CasifySuperTrigger

B.6. Map Demons

6.1 *Map* set_switch

CasifyDemon

1 6.2 *Casify* set_switch

CasifyConjunctiveTrigger

1 6.3 *Map* set_switch_when_bubble_package (sswbp)

UnfoldDemon

2 6.4 *Unfold* sswbp at release_package_into_network

ScatterComputationOfDemon

3 6.5 *Factor* update of packages_due_at_switch

FactorDBMaintenanceIntoAction

1 6.6 *Map* set_switch_on_exit

MapByConsolidation

2 6.7 *Consolidate* set_switch_on_exit and package_leaving_switch

MergeDemons

3 6.8 *Equivalence* triggers

Anchor1

4 6.9 *Reformulate* switch_is_empty as expression

ReformulateDerivedRelation

5 6.10 *Unfold* switch_is_empty in trigger

ScatterComputationOfDerivedRelation

5 6.11 (reposted) *Reformulate* existential as universal

ReformulateExistentialTrigger

6 6.12 *Equivalence* two declarations

Anchor2

6.13 *Map* misrouted_package_reached_bin

CasifyDemon

1 6.14 *Casify* misrouted_package_reached_bin

CasifyConjunctiveTrigger

1 6.15 *Map* misrouted_package_located_at_bin

MapByConsolidation

2 6.16 *Consolidate* misrouted_package_located_at_bin and package_entering_bin

MergeDemons

3 6.17 *Equivalence* declaration lists

EquivalenceCompoundStructures

4 6.18 *Equivalence* bin.reached and bin

Anchor1

4 6.19 (reposted) *Equivalence* declaration lists

AddNewVar

1 6.20 *Map* misrouted_package_destination_set

UnfoldDemon

2 6.21 *Unfold* misrouted_package_destination_set

ScatterComputationOfDemon

Appendix C
Package Router Development

One of the largest and most interesting GIST specifications to date is that of a mechanical package router. The English description of the router is found in section 3.1, and the formal Gist specification in appendix A. Here we present an annotated history of the Glitter development[53]. In this appendix we look at only the goals posted and methods selected; appendix B presents the goal/subgoal structure, appendix D the selection process.

Structure and Notation:

- Development steps. We will present the development as an alternating series of goals and methods for achieving those goals. Goals posted by the user will be underlined and flagged with ***user***, all other goals are generated as a byproduct of problem solving. The goal syntax has been sweetened slightly and abbreviated from the actual menu-driven interaction (see section 2.3.3.2). Noise words have been added for readability. Goals which are trivially satisfied (i.e., hold in the posting state) will generally not be made explicit.

- Program snapshots. Snapshots of the program development state will be given to illustrate the effect of transformations on the specification. The program syntax is described in chapter 3 and appendix A. In some cases, the program will be annotated with $\blacktriangleright_i$s. These will be used as a referencing aid from within the development.

- A large part of the development process can be characterized as information-spreading. Code is introduced by either unfolding or maintaining a particular construct. At intervals during the development it is often useful to regroup by applying simplification transformations which attempt to both get rid of unnecessary buffer code and use the local context to optimize spread code. Simplification is *not* carried out automatically, but must be explicitly invoked through the *Simplify* goal. The timing of the simplification or clean-up intervals is left to the user. They are generally chosen after major surgery has been done to the program. For readability, we have taken some liberties with the timing and

[53] Feather and London have developed a portion of the package router by hand using a transformational approach [London & Feather 82]. While looking at only a portion of the entire development, they provided a large number of insights into the overall development structure.

explicitness of simplification steps: we use them more frequently than is typical and generally only mention that simplification has taken place, leaving the Simplify goal implicit. Because we view the simplification process as below the planning level, we believe this type of omission will make the development easier to follow.

- Trigger/response assumption. We will assume that the response of a demon is executed in the same state that the demon was triggered in. In some cases, this puts implicit constraints on the *environment*, a.k.a. gravity, friction, speed of mechanical sensors. Normally these constraints would show up explicitly as a development progressed; we forego them here for simplicity.

A development digest: For presentation purposes, the development has been sectioned around the user's high level development goals. Below is a synopsis of each section.

1. *Remove* relation PACKAGES_EVER_AT_SOURCE; a moderate task. No need for keeping track of <u>all</u> of the packages that enter the router, just the last one.

2. *Remove* relation PREVIOUS_PACKAGE; a moderate task. Removal of "temporary variable".

3. *Remove* relation LAST_PACKAGE; an easy task. The only information that need be remembered about the last package is its destination.

4. *Map* constraint DID_NOT_SET_SWITCH_WHEN_HAD_CHANCE; a difficult task. Decide switch setting strategy.

5. *Map* relation PACKAGES_DUE_AT_SWITCH; a difficult task. Find way to maintain the fundamental data structure of the system.

6. *Map* demons; a moderate task. Map the demonic structure into triggerings on observable events.

C.1. Remove PACKAGES_EVER_AT_SOURCE

The package router specification provides for keeping the sequence of all packages that ever enter the system in the relation PACKAGES_EVER_AT_SOURCE. However, the only use the spec makes of this relation (sequence) is to access the last package that has entered the system; keeping the entire sequence is wasted overhead. The development will start with the user deciding to remove the unneeded sequence from the specification.

Before proceeding with the development, a note is in order. The process of removing PACKAGES_EVER_AT_SOURCE was the portion of the development studied in detail by Feather and London [London & Feather 82]. A number of the steps in the Feather and London (F&L) development have a Eureka flavor: without an overall explicit development plan, they appear to be pulled out of thin air to allow the development to continue. This is not a criticism of the F&L development in particular. In fact, it was a rather masterful job. Any development which captures only the final set of sequential steps that went into the implementation of a particular spec will naturally be difficult to motivate. Further, a development based on the user searching through a catalog of transformations for a "good" one to apply generally takes the flavor of opportunistic search: 1) try applying a transformation, 2) if it produces something interesting, continue development there, else 3) goto 1. Depending on the complexity of the spec and catalog (expected to be large in both cases), this is not a good model of development. The likelihood of missing either some important step or the right order of step application(found to be a crucial constraint in a TI development) is great. Planning information is clearly needed. The GLITTER development provides an explicit planning structure and succeeds in rationalizing most of the steps; ones remaining unmotivated (i.e., up to the user) are discussed as to their resistance to future automation.

Below is the portion of the spec that we will be working with in this section:

```
demon RELEASE_PACKAGE_INTO_NETWORK(package.new)
  trigger package.new:LOCATED_AT = the source
  response
    begin
      if (the package.previous ||
              package.previous immediately before package.new
▶1                wrt PACKAGES_EVER_AT_SOURCE(*)
          ):DESTINATION ≠ package.new:DESTINATION
       then invoke WAIT[];

      update :LOCATED_AT of package.new to (the source):SOURCE_OUTLET
    end;

relation PACKAGES_EVER_AT_SOURCE(package_seq | sequence of package)
   definition package_seq =
     ({package || (package:LOCATED_AT = the source) asof everbefore}
        ordered temporally by start (package:LOCATED_AT = the source));
```

The initial goal is to get rid of the sequence.

STEP 1.1 *(user)*: *Remove* PACKAGES_EVER_AT_SOURCE from ***spec***[54]

```
| Method  RemoveRelation                                        |

      Goal: Remove R|relation from spec
      Action: 1) forall reference-location[R,RR,spec]
                          do Remove RR from spec
              2) Apply REMOVE_UNREFERENCED_RELATION(R)

      [You can remove a relation if you can remove all references to it.]
| End Method                                                    |
```

In our case, there is only one reference to the sequence: the one ▶$_1$ found in the derived object *package.previous*.

STEP 1.2: *Remove* reference ▶$_1$ to PACKAGES_EVER_AT_SOURCE from *spec*

[54] The entire specification or root of the parse tree.

| Method MegaMove |

Goal: *Remove* X | *relation-reference* from *spec*
Filter: a) component-of[X, Y]
Action: 1) *Isolate* Y in DR | *derived-relation*
2) *MaintainIncrementally* DR

[Remove the relation-reference X by moving it directly after the locations it is assigned.]

| End Method |

Note that the component-of relation is transitive. Hence, a number of different bindings may occur on Y, creating a separate method instantiation for each. The Y we have chosen is the surrounding derived-object. We could have also chosen the more immediate context of the positional-retrieval. In this case, both lead to the same basic state.

STEP 1.3: *Isolate*

```
(the package.previous ||
   package.previous immediately before package.new
      wrt PACKAGES_EVER_AT_SOURCE(*))
```

| Method FoldGenericIntoRelation |

Goal: *Isolate* X
Action: 1) *Globalize* X
2) *Apply* FOLD_INTO_RELATION(X)

[Straightforward fold into derived-relation.]

| End Method |

STEP 1.4: *Globalize*

```
(the package.previous ||
   package.previous immediately before package.new
      wrt PACKAGES_EVER_AT_SOURCE(*))
```

```
| Method  GlobalizeDerivedObject                                  |

        Goal: Globalize DO|derived-object
        Action: 1) forall reference-location[V, $, DO]
                     suchthat V ≠ local-var-of[*, DO]
                      do Try Reformulate V as global-expression

        [Try changing all local variable references to global references.]
| End Method                                                      |
```

Note the use of the **Try** modifier here: each Reformulate goal may be marked as **unrealizable** by the user.

STEP 1.5: Try *Reformulate package.new* (in **derived-object** *package.previous*) as *global-expression*

```
| Method  ReformLocalAsLast                                       |

        Goal: Reformulate V|variable as global-expression
        Filter: a) pattern-match[
                       relation name (seq|sequence of type) def;,
                       R, spec]
                b) domain-type-of[type, V]
        Action: 1) Reformulate V as last(name(*))

        [If you can find a sequence containing the same type of objects as V then you
        may be able to change V into a specific reference to the sequence.]
| End Method                                                      |
```

This method looks for a sequence which is composed of the same type of objects as the variable *package.new*, i.e., the type *package*.

STEP 1.6: *Reformulate package.new* as <u>last</u>(PACKAGES←EVER←AT←SOURCE(*))

At this point, no methods succeed in achieving the goal. The user has two options: 1) since this is part of a try-goal, the user can ignore it and move onto the fold step, or 2) he can manually manipulate the program to achieve the goal. If the latter is chosen, which it is in this

case, the system notes the problem solving context for future (human) analysis; any manual steps taken by the user are assumed to be necessitated by some missing piece of development knowledge in the system. In this case, it is lack of a theorem prover.

STEP 1.7 *(user)*:

Manual MANUAL-REPLACE(*package.new*, last(PACKAGES_EVER_AT_SOURCE(*))

This is the first operation actually carried out in the program space; in the base-line TI system, this would be the first arc of the development path (see the F&L development). Without motivation, i.e., the six subgoals sitting above it, it appears as a somewhat lucky or Eureka step: fortuitously replace an expression with an equivalent value. With the subgoal hierarchy intact, its true purpose is illuminated: prepare the derived-object for isolation (so that it can be maintained so that the reference can be removed ...). Note also the interaction between user and system: the system provides the focusing and motivation while the user is responsible for the deep reasoning necessary to show that the two expressions are equivalent.

After replacing the local with a global expression, we have the following:

```
(the package.previous ||
  package.previous immediately before last(PACKAGES_EVER_AT_SOURCE(*))
     wrt PACKAGES_EVER_AT_SOURCE(*))
```

We now have removed all reliance on local variables (*package.previous* will become the necessary *'ed parameter). If any did remain, the same two options of ignoring the globilization goal (allowing them to become parameters in the newly formed derived relation) or finding a replacement value would be available.

After applying the relation folding transformation FOLD_INTO_RELATION to produce a new relation PREVIOUS_PACKAGE[55] ▸$_1$, we have the following

[55] When the system needs a name for a new item, it asks the user to supply it. User supplied names lead to much more readable programs. With a sophisticated name generating capability, the system might be able to do as well. Currently no such capability exists.

```
demon RELEASE_PACKAGE_INTO_NETWORK(package.new)
  trigger package.new:LOCATED_AT = the source
  response
    begin
      if PREVIOUS_PACKAGE(*):DESTINATION ≠ package.new:DESTINATION
       then invoke WAIT[];
      update :LOCATED_AT of package.new to (the source):SOURCE_OUTLET
    end;

relation PACKAGES_EVER_AT_SOURCE(package_seq | sequence of package)
   definition package_seq =
     ({package || (package:LOCATED_AT = the source) asof everbefore}
        ordered temporally by start (package:LOCATED_AT = the source));

▸1 relation PREVIOUS_PACKAGE(prev_package | package)
   definition prev_package =
    (a package.previous ||
     package.previous immediately < last(PACKAGES_EVER_AT_SOURCE(*))
▸2            wrt PACKAGES_EVER_AT_SOURCE(*));
```

STEP 1.8: *MaintainIncrementally* **PREVIOUS_PACKAGE**

Method ScatterMaintenanceForDerivedRelation

Goal: *MaintainIncrementally* DR|***derived-relation***

Filter: a) ~recursive[DR]

Action: 1) *Flatten* body-of[DR]

2) forall reference-location[BR, S, DR]

```
do forall reference-location[BR, L, spec)
   do begin
        Apply INTRODUCE_MAINTENANCE_CODE(DR L)
        Purify L
      end
```

[To maintain a derived relation DR, find everywhere the base relations of DR are changed and stick code in to maintain. Make sure that all base relations are simple before maintenance and that all code is pure after.]

| **End Method** |

STEP 1.9: *Flatten* **PREVIOUS_PACKAGE**

Flattening the relation body is a simple and inelegant way of insuring that all relations that **PREVIOUS_PACKAGE** relies on are found. A more sophisticated method would attempt to analyze the relation structure to determine the base relation set.

```
| Method  Flatten                                                    |

      Goal: Flatten DR|derived-relation
      Action: 1) forall
            reference-location[BR|derived-relation,$,DR]
                        do Map BR

      [Map all derived relations found in DR into simple ones.]
| End Method                                                         |
```

PACKAGES_EVER_AT_SOURCE $\blacktriangleright_2$ is the only derived relation that is referenced in the PREVIOUS_PACKAGES's definition.

STEP 1.10: *Map* derived-relation PACKAGES_EVER_AT_SOURCE

We have two basic choices in mapping away a derived relation: unfold it everywhere it is used (backward inference); maintain its value at places where its base information changes (forward inference). We have chosen the latter.

```
| Method  MaintainDerivedRelation                                    |

      Goal: Map DR|derived-relation
      Action: 1) MaintainIncrementally DR

      [One way of mapping a derived relation is to maintain it explicitly.]
| End Method                                                         |
```

STEP 1.11: *MaintainIncrementally* PACKAGES_EVER_AT_SOURCE

| Method IntroduceSeqMaintenanceDemon |

Goal: *MaintainIncrementally* DR | ***derived-relation***

Filter: a) gist-type-of[parameter-of[DR], ***sequence***]

Action: 1) *Reformulate* body-of[DR] as **temporally-ordered-set-idiom**[56]

2) *Apply* INTRODUCE_SEQ_MAINTENANCE_DEMON(DR)

[One way of maintaining a derived sequence is to first change the definition into a temporal order -- ({x||P(x)asof everbefore} ordered temporally by P(x)) -- and then set up a demon with trigger P(x) to add elements.]

| End Method |

The relation PACKAGES_EVER_AT_SOURCE is already in the desired form, so a new demon is introduced, NOTICE_NEW_PACKAGE_AT_SOURCE $\blacktriangleright_1$, to add packages to the sequence when they arrive at the source:

[56] Patterns can be predefined and named. In this case, ({x||P(x) asof everbefore} ordered temporally by start P(x)).

```
demon RELEASE_PACKAGE_INTO_NETWORK(package.new)
  trigger package.new:LOCATED_AT = the source
  response
    begin
      if PREVIOUS_PACKAGE(*):DESTINATION ≠ package.new:DESTINATION
       then invoke WAIT[];

      update :LOCATED_AT of package.new to (the source):SOURCE_OUTLET
    end;

relation PACKAGES_EVER_AT_SOURCE(package_seq | sequence of package);

relation PREVIOUS_PACKAGE(prev_package | package)
   definition prev_package =
    (a package.previous ||
     package.previous immediately before last(PACKAGES_EVER_AT_SOURCE(*))
          wrt PACKAGES_EVER_AT_SOURCE(*));

▸1 demon NOTICE_NEW_PACKAGE_AT_SOURCE(package)
  trigger package:LOCATED_AT = the source
  response
▸2   update package_seq in PACKAGES_EVER_AT_SOURCE($)
          to PACKAGES_EVER_AT_SOURCE(*) concat <package>;
```

Having flattened PREVIOUS_PACKAGE's body, we are now ready to maintain it by finding all the places its base information (i.e., PACKAGES_EVER_AT_SOURCE) changes. There is only one place to worry about: the update of PACKAGES_EVER_AT_SOURCE ▸$_2$ in the demon NOTICE_NEW_PACKAGE_AT_SOURCE. After applying the maintenance transformation INTRODUCE_MAINTENANCE_CODE, the program is as follows:

```
demon RELEASE_PACKAGE_INTO_NETWORK(package.new)
  trigger package.new:LOCATED_AT = the source
  response
    begin
      if PREVIOUS_PACKAGE(*):DESTINATION ≠ package.new:DESTINATION
       then invoke WAIT[];
      update :LOCATED_AT of package.new to (the source):SOURCE_OUTLET
    end;

relation PACKAGES_EVER_AT_SOURCE(package_seq | sequence of package);

relation PREVIOUS_PACKAGE(prev_package | package);

demon NOTICE_NEW_PACKAGE_AT_SOURCE(package)
  trigger package:LOCATED_AT = the source
  response
    atomic
       update package_seq in PACKAGES_EVER_AT_SOURCE($)
         to PACKAGES_EVER_AT_SOURCE concat <package>;
       update prev_package in PREVIOUS_PACKAGE($)
         to (the package.previous ||
                 package.previous immediately before
                   last(PACKAGES_EVER_AT_SOURCE(*) concat <package>)
                    wrt PACKAGES_EVER_AT_SOURCE(*) concat <package>)
    end atomic
```

Our next goal is the purification of NOTICE_NEW_PACKAGE_AT_SOURCE: if that demon is not within our portion of the development then we must move the newly introduced code out of it and into our portion. In this case, we have defined the demon as part of our portion so the goal is trivially satisfied.

We have now achieved our goal of maintaining the derived relation PREVIOUS_PACKAGE. Further, the MegaMove method used to remove the sole reference to PACKAGES_EVER_AT_SOURCE has completed. However, the reference has not been eliminated, but simply moved. As described in chapter 5, this causes the remove goal from step 1.2 to be re-activated[57]. The system automatically keeps track of the movement of the reference in order to update the arguments of remove:

[57] This is equivalent to a recursive posting of a Remove goal as the last action of MegaMove.

STEP 1.12: *Remove* reference of PACKAGES_EVER_AT_SOURCE in

```
(the package.previous ||
    package.previous immediately before
        last(PACKAGES_EVER_AT_SOURCE(*) concat <package>)
          wrt PACKAGES_EVER_AT_SOURCE(*) concat <package>)
from spec
```

Using MegaMove again will lose: PREVIOUS_PACKAGE (under another name) will simply be re-introduced. We will try a different approach. It is often the case that when dealing with a sequence, it is easier to manipulate a positional retrieval (e.g., first, last, Nth) than a relative one (e.g., (immediately) before, (immediately) after). The method we will employ involves reformulating the relative retrieval into a positional one and then trying MegaMove on that.

```
| Method  PositionalMegaMove                                          |

      Goal:   Remove RR|relation-reference from spec
      Filter: a) RR component-of Y
      Action: 1) Reformulate Y as PR|positional-retrieval
              2) Isolate PR in DR|derived-relation
              3) MaintainIncrementally DR

      [One way of getting rid of a reference to a sequence is to reformulate it as part
      of a positional retrieval, and then megamove it.]
| End Method                                                          |
```

As is usual, the binding we choose for Y is important. In this case it is the entire derived object. The development from this point involves several low level reformulation steps. Note that without the rich teleology provided by Glitter,these steps in particular and low level steps in general are hard to motivate and often appear fortuitous in a base-line development (see for instance [London & Feather 82]).

STEP 1.13: *Reformulate*

```
(the package.previous ||
      package.previous immediately before
        last(PACKAGES_EVER_AT_SOURCE(*) concat <package>)
         wrt PACKAGES_EVER_AT_SOURCE(*) concat <package>)

   as positional-retrieval
```

```
| Method  ReformulateDerivedObject                                    |

        Goal:   Reformulate DO|derived-object as P
        Action: 1) Reformulate body-of[DO]
                         as local-var-of[*, DO]=P
                2) Apply UNFOLD_DERIVED_OBJECT(DO)

        [(x || x = P) => P]
| End Method                                                          |
```

P is bound to the abstract type ***positional-retrieval.*** Our new goal is to reformulate the body of the derived object into a equivalence relation involving the free variable *package.previous* and a (any) positional-retrieval.

STEP 1.14: *Reformulate*

```
package.previous immediately before
  last(PACKAGES_EVER_AT_SOURCE(*) concat <package>)
    wrt PACKAGES_EVER_AT_SOURCE(*) concat <package>)

    as package.previous=positional-retrieval
```

```
| Method  ReformulateRelativeRetrievalAsLast                          |

        Goal:   Reformulate RS|relative-sequence-retrieval
                             as "x|object=last(Seq|SEQUENCE)"
        Action: 1) Reformulate RS as
                     "x immediately before y wrt (Seq concat z)"
                2) Equivalence y and z
                3) Apply CHANGE_TO_RETRIEVAL_OF_LAST(RS)

        [x immediately before y wrt (Seq concat y) => x = last(Seq)]
| End Method                                                          |
```

Note that the above method's trigger will match **positional-retrieval**, the more general goal pattern, with last(Seq), the more specific pattern required by the method. Naturally, there will be a competing method to the above that attempts to reformulate to first(Seq).

The reformulation goal is trivially satisfied: the program matches in the current state. However, we must equivalence y and z.

STEP 1.15: *Equivalence*

```
last(PACKAGES_EVER_AT_SOURCE(*) concat package)
and
package
```

```
| Method  Anchor2                                                        |

     Goal: Equivalence X and Y
     Action: 1) Reformulate X as Y

     [Try changing the first construct into something that matches the second.]
| End Method                                                             |
```

STEP 1.16: *Reformulate*

```
last(PACKAGES_EVER_AT_SOURCE(*) concat package)
as package
```

```
| Method  ReformulateAsObject                                            |

     Goal: Reformulate SR|last-retrieval as O|object
     Action: 1) Reformulate parameter-of[*, SR] as
                    (S concat O)
             2) Apply SIMPLIFY_LAST(SR)

     [last(Seq concat O) => O]
| End Method                                                             |
```

The Reformulation goal is trivially satisfied. At this point, we are ready to unwind the nested goals we have built up. After application of SIMPLIFY_LAST we have:

```
(the package.previous ||
     package.previous immediately before package
       wrt PACKAGES_EVER_AT_SOURCE(*) concat <package>)
```

After application of CHANGE_TO_RETRIEVAL_OF_LAST we have:

(the *package.previous* ||
package.previous = last(PACKAGES_EVER_AT_SOURCE(*))

After applying transformation UNFOLD_DERIVED_OBJECT we have:

update *prev_package* in PREVIOUS_PACKAGE($)
to last(PACKAGES_EVER_AT_SOURCE(*))

The reformulation necessary in this portion of the development is caused by the fussiness of the development methods we employ. All of the above reformulation could be eliminated if we wished to include a method which looks specifically for the following case:

```
(x || x immediately before last(s concat z)
        wrt (s concat z)).
```

Such a method could directly reformulate the derived object. Of course, we would need an infinite number of such methods to cover all of the possible cases.

We are now ready to isolate the retrieval of PACKAGES_EVER_AT_SOURCE.

STEP 1.17: *Isolate* last(PACKAGES_EVER_AT_SOURCE(*))

| Method FoldGenericIntoRelation |

Goal: *Isolate* X

Action: 1) *Globalize* X

2) *Apply* FOLD_INTO_RELATION(X)

[Straightforward fold into derived-relation.]

| End Method |

There are no local variables in the action to be isolated, hence the Globalize goal is trivially satisfied. Application of FOLD_INTO_RELATION results in the introduction of a new derived relation $\blacktriangleright_2$:

```
demon RELEASE_PACKAGE_INTO_NETWORK(package.new)
  trigger package.new:LOCATED_AT = the source
  response
    begin
      if PREVIOUS_PACKAGE(*):DESTINATION ≠ package.new:DESTINATION
       then invoke WAIT[];

      update :LOCATED_AT of package.new to (the source):SOURCE_OUTLET
    end;

relation PACKAGES_EVER_AT_SOURCE(package_seq | sequence of package);

relation PREVIOUS_PACKAGE(prev_package | package);

demon NOTICE_NEW_PACKAGE_AT_SOURCE(package)
  trigger package:LOCATED_AT = the source
  response
    atomic
▸1      update package_seq in PACKAGES_EVER_AT_SOURCE($)
            to PACKAGES_EVER_AT_SOURCE concat <package>;
       update prev_package in PREVIOUS_PACKAGE($)
            to LAST_PACKAGE(*)
    end atomic;

▸2 relation LAST_PACKAGE(last_package | package)
     definition last_package = last(PACKAGES_EVER_AT_SOURCE);
```

STEP 1.18: *MaintainIncrementally* LAST_PACKAGE

We will use the same method here to maintain LAST_PACKAGE that we used earlier to maintain PREVIOUS_PACKAGE:

| **Method** ScatterMaintenanceForDerivedRelation |

```
Goal: MaintainIncrementally DR|derived-relation
Action: 1) Flatten body-of[DR]
        2) forall reference-location[BR, S, DR]
         do forall reference-location[BR, L, spec)
               do begin
                 Apply INTRODUCE_MAINTENANCE_CODE(DR L)
                 Purify L
               end
```

[To maintain a derived relation DR, find everywhere the base relations of DR are changed and stick code in to maintain. Make sure that all base relations are simple before maintenance and that all code is pure after.]

| **End Method** |

The Flatten goal is trivially satisfied. After application of the INTRODUCE_MAINTENANCE_CODE transformation at the sole place where **PACKAGES_EVER_AT_SOURCE** is changed $\blacktriangleright_2$, we have the following state:

```
demon RELEASE_PACKAGE_INTO_NETWORK(package.new)
  trigger package.new:LOCATED_AT = the source
  response
    begin
      if PREVIOUS_PACKAGE(*):DESTINATION ≠ package.new:DESTINATION
       then invoke WAIT[];
      update :LOCATED_AT of package.new to (the source):SOURCE_OUTLET
    end;

relation PACKAGES_EVER_AT_SOURCE(package_seq | sequence of package);

relation PREVIOUS_PACKAGE(prev_package | package);

demon NOTICE_NEW_PACKAGE_AT_SOURCE(package)
  trigger package:LOCATED_AT = the source
  response
    atomic
       update package_seq in PACKAGES_EVER_AT_SOURCE($)
            to PACKAGES_EVER_AT_SOURCE concat <package>;
       update prev_package in PREVIOUS_PACKAGE($)
            to LAST_PACKAGE(*);
▸1      update last_package in LAST_PACKAGE($)
            to last(PACKAGES_EVER_AT_SOURCE(*) concat <package>)
    end atomic;

relation LAST_PACKAGE(last_package | package);
```

The MegaMove method has completed and we still have not gotten rid of the reference of PACKAGES_EVER_AT_SOURCE. However, we are fairly close now. The Remove goal is re-activated:

STEP 1.19: *Remove* reference of PACKAGES_EVER_AT_SOURCE in ▸$_1$ from ***spec***

Our previous strategy has been to isolate/maintain (a.k.a. MegaMove) references of the sequence. At this point, we have enough information to try a new tact: replace the sequence reference by an actual object.

```
| Method  RemoveByObjectizingContext                              |

        Goal: Remove RR | relation-reference from spec
        Filter: a) component-of[RR, Y]
        Action: 1) Reformulate Y as object

        [One way of getting rid of a relation reference which is embedded in context Y
        is to reformulate Y as an explicit object.]
| End Method                                                      |
```

Here we bind Y to the most immediate context of the reference, the positional retrieval <u>last</u>.

STEP 1.20: *Reformulate*

```
        last(PACKAGES_EVER_AT_SOURCE(*) concat <package>)
     as object
```

Using the same method as in step 1.15, **ReformulateAsObject**, we get the following:

```
demon RELEASE_PACKAGE_INTO_NETWORK(package.new)
  trigger package.new:LOCATED_AT = the source
  response
    begin
      if PREVIOUS_PACKAGE(*):DESTINATION ≠ package.new:DESTINATION
       then invoke WAIT[];
      update :LOCATED_AT of package.new to (the source):SOURCE_OUTLET
    end;

relation PACKAGES_EVER_AT_SOURCE(package_seq | sequence of package);

relation PREVIOUS_PACKAGE(prev_package | package);

demon NOTICE_NEW_PACKAGE_AT_SOURCE(package)
  trigger package:LOCATED_AT = the source
  response
    atomic
▸2       update package_seq in PACKAGES_EVER_AT_SOURCE($)
              to PACKAGES_EVER_AT_SOURCE concat <package>;
        update prev_package in PREVIOUS_PACKAGE($)
              to LAST_PACKAGE(*);
        update last_package in LAST_PACKAGE($)
              to package
    end atomic;

relation LAST_PACKAGE(last_package | package);
```

Note that this last step is traditionally viewed as *simplification* steps which are automatically applied whenever possible, e.g., last(S concat X) ⇒ X (see [Standish et al 76], [Rutter 77]). These type of steps have the weakest connection to the rest of the development. They appear to be independent and opportunistic. Here, we strongly tie in the "simplification" as a necessary step in the higher level goal of removing the need for the sequence PACKAGES_EVER_AT_SOURCE.

We have one remaining reference to PACKAGES_EVER_AT_SOURCE ▸$_2$ that we must remove:

STEP 1.21: *Remove*

```
        update package_seq in PACKAGES_EVER_AT_SOURCE($)
            to PACKAGES_EVER_AT_SOURCE concat <package>
 from spec
```

```
| Method  RemoveUnusedAction                                        |

      Goal: Remove A|action
      Action: 1) Show action_is_unnoticed(A)
              2) Apply REMOVE-UNNOTICED-ACTION(A)

      {Show that the current action is either not used or superseded by a
      subsequent action.}
| End Method                                                        |
```

STEP 1.22: *Show* **action_is_unnoticed(**

update *package_seq* **in** **PACKAGES_EVER_AT_SOURCE($)**
to **PACKAGES_EVER_AT_SOURCE** **concat** *<package>***)**

```
| Method  ShowDysteleological                                       |

      Goal: Show action_is_unnoticed(U|update)
      Filter: a) update-relation-of[R, U]
              b) ~reference-location[R, $, spec]
      Action: 1) Assert action_is_unnoticed(U)

      [If you are trying to show that an update is unnoticed, show that it is never
      referenced.]
| End Method                                                        |
```

Since there are no references to PACKAGES_EVER_AT_SOURCE, we can assert that it is unnoticed. After removal of the update and the relation definition, we have the following (In an unstructured development, the removal here of the PACKAGES_EVER_AT_SOURCE sequence might appear as a fortunate and opportunistic by-product of the preceding steps. Here, it is just one step (the last) of a general plan aimed at getting rid of the sequence.):

```
demon RELEASE_PACKAGE_INTO_NETWORK(package.new)
  trigger package.new:LOCATED_AT = the source
  response
    begin
      if PREVIOUS_PACKAGE(*):DESTINATION ≠ package.new:DESTINATION
       then invoke WAIT[];
      update :LOCATED_AT of package.new to (the source):SOURCE_OUTLET
    end;

▶1 relation PREVIOUS_PACKAGE(prev_package | package);

▶2 demon NOTICE_NEW_PACKAGE_AT_SOURCE(package)
  trigger package:LOCATED_AT = the source
  response
    atomic
       update prev_package in PREVIOUS_PACKAGE($)
            to LAST_PACKAGE(*);
       update last_package in LAST_PACKAGE($)
            to package
    end atomic;

▶3 relation LAST_PACKAGE(last_package | package);
```

This completes the removal of the PACKAGES_EVER_AT_SOURCE relation. However, a new demon ▶$_2$ and two new relations ▶$_1$,▶$_3$ have been introduced as side-effects of the removal process. The next two sections deal with further developing and optimizing these components.

C.2. Remove PREVIOUS_PACKAGE

The next portion of the development involves noticing that PREVIOUS_PACKAGE is acting as a temporary variable for LAST_PACKAGE.

```
demon NOTICE_NEW_PACKAGE_AT_SOURCE(package)
  trigger package:LOCATED_AT = the source
  response
    atomic
▸1     update prev_package in PREVIOUS_PACKAGE($)
            to LAST_PACKAGE(*);
▸2     update last_package in LAST_PACKAGE($)
            to package
    end atomic;

demon RELEASE_PACKAGE_INTO_NETWORK(package.new)
  trigger package.new:LOCATED_AT = the source
  response
    begin
▸3    if PREVIOUS_PACKAGE(*):DESTINATION ≠ package.new:DESTINATION
       then invoke WAIT[];
      update :LOCATED_AT of package.new to (the source):SOURCE_OUTLET
    end;

relation PREVIOUS_PACKAGE(prev_package | package);

relation LAST_PACKAGE(last_package | package);
```

The general pattern, if we wanted to do this noticing automatically is

```
X <- Y;
Y <- c;
E|expression using X
```

This matches the following code, where X is bound to PREVIOUS←PACKAGE, Y bound to LAST←PACKAGE and E to the conditional wait ▸3.

```
    atomic
▸1     update prev_package in PREVIOUS_PACKAGE($)
          to LAST_PACKAGE(*);
▸2     update last_package in LAST_PACKAGE($)
          to package.new
    end atomic;
    ...
▸3  if PREVIOUS_PACKAGE(*):DESTINATION ≠ package.new:DESTINATION
     then invoke WAIT[];
```

We can generally get rid of the need for X (PREVIOUS_PACKAGE) by computing consecutively the assignment of X with its use (the conditional wait $▸_3$) and replacing X with Y (LAST_PACKAGE).

STEP 2.1 *(user)*: *Remove* PREVIOUS_PACKAGE

```
| Method  RemoveRelation                                        |

      Goal: Remove R|relation from spec
      Action: 1) forall reference-location[R,RR,spec]
                         do Remove RR from spec
              2) Apply REMOVE_UNREFERENCED_RELATION(R)

      [You can remove a relation if you can remove all references to it.]
| End Method                                                    |
```

STEP 2.2: *Remove* reference of PREVIOUS_PACKAGE in $▸_3$ from *spec*

```
| Method  ReplaceRefWithValue                                   |

      Goal: Remove R|simple-relation-reference
      Action: 1) Show VALUE_KNOWN(R, V)
              2) Apply REPLACE_REF_WITH_VALUE(R V)

      [One way of getting rid of a relation reference is to replace it with its value.]
| End Method                                                    |
```

Note that another competing method here is **MegaMove**. That is, we could isolate the reference PREVIOUS_PACKAGE(*):DESTINATION into a new derived-relation and then

maintain it. However, this has the negative effect of introducing still another temporary variable (relation). While we can get rid of this too eventually, the process will be messier. In general, a method which removes a reference by replacing it with a value is preferred over a method which replaces it (or its surroundings) with another reference.

STEP 2.3: *Show* VALUE_KNOWN(PREVIOUS_PACKAGE(*), V)

```
| Method  ShowUpdateGivesValue                                        |

        Goal:   Show VALUE_KNOWN(R | relation-reference, V)
        Filter: a) pattern-match[update, U, spec]
                b) name-of[R] = update-relation-of[*, U]
        Action: 1) Show UPDATE_VALUE_HOLDS(U, R)
                2) Assert VALUE_KNOWN(R, new-value-of[*, U])

        [Find the last update of R and show that the new value is still valid.]
| End Method                                                          |
```

There is only one update of PREVIOUS_PACKAGE in the spec, the one found in NOTICE←NEW←PACKAGE←AT←SOURCE. We now must show that the value the relation was set to is still around.

STEP 2.4: *Show*

LAST_PACKAGE(*) (in ▸$_1$)

still holds at

▸$_3$ <u>if</u> PREVIOUS_PACKAGE(*):DESTINATION ≠ *package.new*:DESTINATION
<u>then</u> <u>invoke</u> WAIT[];

```
| Method  ShowNewValueStillValid                                    |

        Goal: Show UPDATE_VALUE_HOLDS(U|update,
                          R|relation reference)
        Filter: a) name-of[R] = update-relation-of[*, U]
        Action: 1) Show
       UNCHANGED_BETWEEN_LOCATIONS(new-value-of[*, U], U, R)
                3) Assert  UPDATE_VALUE_HOLDS(U, R)

        [To show that the new update value is still around at R, show that the update
        value has not been changed before R.]
| End Method                                                        |
```

STEP 2.5: *Show* LAST_PACKAGE doesn't change between ▶$_1$ and ▶$_3$.

```
| Method  MoveInterveningUpdate                                     |

        Goal: Show UNCHANGED_BETWEEN_LOCATIONS(V|relation reference,
                                               U|update,
                                               R|relation reference)
        Filter: a) pattern-match[update, L, spec]
                b) update-relation-of[V, L]
        Action: 1) Show COMPUTATIONALLY-BETWEEN[L, U, R]
                2) ComputeSequentially R before L

        [If an intervening update of V exists, move it after R.]
| End Method                                                        |
```

In this case, there does exist an intervening update ▶$_2$ to V (LAST_PACKAGE), and hence we will try to move it after ▶$_3$.

STEP 2.6: *ComputeSequentially*

```
▶3      if PREVIOUS_PACKAGE(*):DESTINATION neq package.new:DESTINATION
        then invoke WAIT[];
   before
▶2         update last_package in LAST_PACKAGE($)
              to package.new
```

```
| Method  MoveOutOfAtomic                                        |

     Goal: ComputeSequentially B|action before A|action
     Filter: a) component-of[A, C|atomic]
     Action: 1) Unfold C

     [If you are trying to move A after B and A is in an atomic, unfold the atomic
     before attempting to continue.]
| End Method                                                     |
```

STEP 2.7: *Unfold*

```
atomic
   update prev_package in PREVIOUS_PACKAGE($)
        to LAST_PACKAGE(*);
   update last_package in LAST_PACKAGE($)
        to package
end atomic;
```

```
| Method  UnfoldAtomic                                           |

     Goal: Unfold A|atomic
     Action: 1) Show SEQUENTIAL-ORDERING(O|ordering, A)
             2) Show SUPERFLUOUS_ATOMIC(A)
             3) Apply UNFOLD-ATOMIC(A, O)

     [You can unfold an atomic if you can show that there exists some valid
     sequential ordering of the statements and that no demonic or inferencing
     processes will be effected.]
| End Method                                                     |
```

Currently the user is required to show both of the properties. In the particular case at hand, it would not be difficult to define a method for ordering the statements using a data-dependency graph, something Glitter presently does not have. Showing that the atomic is actually superfluous will probably remain the user's responsibility for some time to come.

After unfolding, the program is as follows:

```
demon NOTICE_NEW_PACKAGE_AT_SOURCE(package)
  trigger package:LOCATED_AT = the source
  response
    begin
▶1     update prev_package in PREVIOUS_PACKAGE($)
            to LAST_PACKAGE(*);
▶2     update last_package in LAST_PACKAGE($)
            to package
    end;

demon RELEASE_PACKAGE_INTO_NETWORK(package.new)
  trigger package.new:LOCATED_AT = the source
  response
    begin
▶3    if PREVIOUS_PACKAGE(*):DESTINATION ≠ package.new:DESTINATION
       then invoke WAIT[];
      update :LOCATED_AT of package.new to (the source):SOURCE_OUTLET
    end;

relation PREVIOUS_PACKAGE(prev_package | package);

relation LAST_PACKAGE(last_package | package);
```

STEP 2.8 ***(reposted)*****:** *ComputeSequentially*

```
▶3     if PREVIOUS_PACKAGE(*):DESTINATION neq package.new:DESTINATION
       then invoke WAIT[];

   before

▶2        update last_package in LAST_PACKAGE($)
              to package.new
```

```
| Method  ConsolidateToMakeSequential                          |

       Goal: ComputeSequentially A1|action before A2|action
       Filter: a) component-of[A1, D1|demon]
       Action: 1) Consolidate D1 and D2

       [It is easier to move actions around if they are in the same context.]
| End Method                                                   |
```

STEP 2.9: *Consolidate*

NOTICE_NEW_PACKAGE_AT_SOURCE and RELEASE_PACKAGE_INTO_NETWORK

```
| Method  MergeDemons                                              |

        Goal: Consolidate D1|demon and D2|demon
        Action: 1) Equivalence trigger-of[D1] and
                                 trigger-of[D2]
                2) Equivalence var-declaration-of[D1] and
                                 var-declaration-of[D2]
                3) Show MERGEABLE_DEMONS(D1, D2, I|ordering)
                4) Apply DEMON_MERGE(D1, D2, I)

        [You can consolidate two demons if you can show that they have the same
        local variables, the same triggering pattern and that they meet certain
        merging conditions.]
| End Method                                                       |
```

STEP 2.10: *Equivalence* (*package.new*) and (*package*)

```
| Method  EquivalenceCompoundStructures2                           |

        Goal: Equivalence  S1|compound-structure and
                           S2|compound-structure
        Filter: a) gist-type-of[*, S1] = gist-type-of[*, S2]
                b) ~fixed-structure[S1]
                c) component-correspondence[S1, S2, C|correspondence]
        Action: 1) forall correspondence-pairs[C, C1, C2]
                     do Equivalence C1 and C2

        {Divide-and-conquer: make the components of two non-fixed structures
        equivalent.}
| End Method                                                       |
```

EquivalenceCompoundStructures2 will compute a correspondence between the variables in the list (in this case only one exists) and post an equivalence goal pair.

STEP 2.11: *Equivalence package* and *package.new*

We can use the brother of method Anchor2 (see step 1.15) to achieve the Equivalence goal here.

```
| Method  Anchor1                                              |

        Goal: Equivalence X and Y
        Action: 1) Reformulate Y as X

        [Try changing the second construct into something that matches the first.]
| End Method                                                   |
```

STEP 2.12: *Reformulate package as package.new*

The achievement of this goal rests on the renaming of *package* to *package.new* within NOTICE←NEW←PACKAGE←AT←SOURCE.

```
| Method  RenameVar                                            |

        Goal: Reformulate V1|variable-declaration as
                                  V2|variable-declaration
        Filter: a) scoped-in[V1 S]
        Action: 1) Show INTRODUCEABLE←VAR←NAME(V2, S)
                2) Apply RENAME_VAR(V1, V2, S)

        [Replace all occurrences of V1 with V2 in S after showing that V2 does not
        conflict with scoped variables already defined within S.]
| End Method                                                   |
```

We assume that the user verifies that the introduction of *package.new* does not conflict with any existing variables within NOTICE←NEW←PACKAGE←AT←SOURCE. After the renaming, the equivalence goal on the triggers is trivially satisfied. The application of DEMON_MERGE gives us

```
demon RELEASE_PACKAGE_INTO_NETWORK(package.new)
  trigger package.new:LOCATED_AT = the source
  response
    begin
▸1    update prev_package in PREVIOUS_PACKAGE($)
             to LAST_PACKAGE(*);
▸2    update last_package in LAST_PACKAGE($)
             to package.new
▸3    if PREVIOUS_PACKAGE(*):DESTINATION ≠ package.new:DESTINATION
       then invoke WAIT[];
      update :LOCATED_AT of package.new to (the source):SOURCE_OUTLET
    end;

relation PREVIOUS_PACKAGE(prev_package | package);

relation LAST_PACKAGE(last_package | package);
```

The *ComputeSequentially* goal from 2.8 is still not satisfied and hence, is reposted.

STEP 2.13 ***(reposted)*****:** *ComputeSequentially*

```
▸3     if PREVIOUS_PACKAGE(*):DESTINATION neq package.new:DESTINATION
       then invoke WAIT[];

    before

▸2        update last_package in LAST_PACKAGE($)
              to package.new
```

| Method   SwapUp |

Goal: *ComputeSequentially* Y before X

Filter: a) brother-of[X, Y]

Action: 1) Swap Y with predecessor of Y

[If you are trying to compute X after Y then move Y up.]

| End Method |

STEP 2.14: *Swap*

```
▸3      if PREVIOUS_PACKAGE(*):DESTINATION ≠ package.new:DESTINATION
          then invoke WAIT[];
 with
▸2      update last_package in LAST_PACKAGE($)
            to package.new;
```

```
| Method  SwapStatements                                    |

        Goal: Swap A with B
        Action: 1) Show SWAPPABLE(A B)
                2) Apply SWAP_STATEMENTS(A B)

        [A;B ⇒ B;A under certain conditions.]
| End Method                                                |
```

Again, with a data-dependency graph, the SWAPPABLE property might automatically be verified. Currently, we rely on the user to verify it. After applying the swap transformation, we have:

```
     begin
▸1      update prev_package in PREVIOUS_PACKAGE($)
            to LAST_PACKAGE(*);
▸3      if PREVIOUS_PACKAGE(*):DESTINATION ≠ package.new:DESTINATION
          then invoke WAIT[];
▸2      update last_package in LAST_PACKAGE($)
            to package.new
       update :LOCATED_AT of package.new to (the source):SOURCE_OUTLET
     end;
```

The ComputeSequentially goal has now been satisfied. After the application of the value replacement transformation REPLACE_REF_WITH_VALUE and the removal of the maintenance and definition (see steps 1.20 and 1.21) of PREVIOUS_PACKAGE, we have:

```
demon RELEASE_PACKAGE_INTO_NETWORK(package.new)
  trigger package.new:LOCATED_AT = the source
  response
    begin
      if LAST_PACKAGE(*):DESTINATION ≠ package.new:DESTINATION
         then invoke WAIT[];
      update last_package in LAST_PACKAGE($)
           to package.new
      update :LOCATED_AT of package.new to (the source):SOURCE_OUTLET
    end;

relation LAST_PACKAGE(last_package | package);
```

This completes the removal of PREVIOUS←PACKAGE.

C.3. Remove LAST_PACKAGE

The next portion of the development involves noticing that we don't need to remember the last package, but only its :DESTINATION ▸$_1$. We might expect an automatic usage analysis to point out such features of the program. Such an analysis is certainly state-of-the-art and should be one of the more immediate enhancements to the TI system.

```
demon RELEASE_PACKAGE_INTO_NETWORK(package.new)
  trigger package.new:LOCATED_AT = the source
  response
    begin
▸1     if LAST_PACKAGE(*):DESTINATION ≠ package.new:DESTINATION
         then invoke WAIT[];
      update last_package in LAST_PACKAGE($)
           to package.new
      update :LOCATED_AT of package.new to (the source):SOURCE_OUTLET
    end;

relation LAST_PACKAGE(last_package | package);
```

Note that remembering all of an objects attributes instead of the object itself may not payoff in cases where a large number of the object's attributes are needed: we may simply be replacing a central "record" structure (an object and its attributes) with individual variables (the isolated relations). In our case, only one field is ever needed, and hence we can perceive an efficiency gain.

<u>STEP 3.1</u> *(user)*: *Remove* LAST_PACKAGE

We will employ the same general "MegaMove" strategy as used in removing the PACKAGES_EVER_AT_SOURCE in section C.1.

| Method RemoveRelation |

Goal: *Remove* R|*relation* from *spec*

Action: 1) forall reference-location[R,RR,*spec*]
do *Remove* RR from *spec*
2) *Apply* REMOVE_UNREFERENCED_RELATION(R)

[You can remove a relation if you can remove all references to it.]

| End Method |

STEP 3.2: *Remove* reference of LAST_PACKAGE in $\blacktriangleright_1$

| Method MegaMove |

Goal: *Remove* X|*relation-reference* from *spec*

Filter: a) component-of[X, Y]

Action: 1) *Isolate* Y in DR|*derived-relation*
2) *MaintainIncrementally* DR

[Remove the relation-reference X by moving it directly after the locations it is assigned.]

| End Method |

We choose the binding of Y as LAST_PACKAGE(*):DESTINATION.

STEP 3.3: *Isolate* LAST_PACKAGE(*):DESTINATION

| Method FoldGenericIntoRelation |

Goal: *Isolate* X

Action. 1) *Globalize* X
2) *Apply* FOLD_INTO_RELATION(X)

[Straightforward fold into derived-relation.]

| End Method |

After applying FOLD_INTO_RELATION, we have:

```
demon RELEASE_PACKAGE_INTO_NETWORK(package.new)
  trigger package.new:LOCATED_AT = the source
  response
    begin
      if LAST_PACKAGE_DESTINATION(*) ≠ package.new:DESTINATION
        then invoke WAIT[];
      update last_package in LAST_PACKAGE($)
           to package.new
      update :LOCATED_AT of package.new to (the source):SOURCE_OUTLET
    end;

relation LAST_PACKAGE(last_package | package);

relation LAST_PACKAGE_DESTINATION(last_destination | bin)
      definition last_destination = LAST_PACKAGE(*):DESTINATION;
```

STEP 3.4: *MaintainIncrementally* **LAST_PACKAGE_DESTINATION**

```
| Method  ScatterMaintenanceForDerivedRelation                    |

        Goal: MaintainIncrementally DR|derived-relation
        Action: 1) Flatten body-of[DR]
                2) forall reference-location[BR, $, DR]
                 do forall reference-location[BR, L, spec)
                      do begin
                        Apply INTRODUCE_MAINTENANCE_CODE(DR L)
                        Purify L
                       end

        [To maintain a derived relation DR, find everywhere the base relations of DR
        are changed and stick code in to maintain. Make sure that all base relations
        are simple before maintenance and that all code is pure after.]
| End Method                                                      |
```

The Flatten goal is trivially satisfied. After adding the necessary maintenance code ▸$_2$, we have:

```
demon RELEASE_PACKAGE_INTO_NETWORK(package.new)
  trigger package.new:LOCATED_AT = the source
  response
    begin
      if LAST_PACKAGE_DESTINATION(*) ≠ package.new:DESTINATION
         then invoke WAIT[];
      atomic
▸1         update last_package in LAST_PACKAGE($)
                to package.new;
▸2         update last_destination in LAST_PACKAGE_DESTINATION($)
                to package.new:DESTINATION
      end atomic

      update :LOCATED_AT of package.new to (the source):SOURCE_OUTLET
    end;

relation LAST_PACKAGE(last_package | package);

relation LAST_PACKAGE_DESTINATION(last_destination | bin);
```

We have now achieved our goal of removing one of the references to LAST_PACKAGE. The next reference ▸$_1$ is part of the maintenance/update of LAST_PACKAGE.

STEP 3.5: *Remove* reference to LAST_PACKAGE from ▸$_1$

We will omit the steps here of removing this reference and the relation definition. They are completely analogous to the steps found at step 1.20-1,21. Our new state is

```
demon RELEASE_PACKAGE_INTO_NETWORK(package.new)
  trigger package.new:LOCATED_AT = the source
  response
    begin
      if LAST_PACKAGE_DESTINATION(*) ≠ package.new:DESTINATION
        then invoke WAIT[];
▸3     atomic
      update last_destination in LAST_PACKAGE_DESTINATION($)
           to package.new:DESTINATION
     end atomic
      update :LOCATED_AT of package.new to (the source):SOURCE_OUTLET
    end;

relation LAST_PACKAGE_DESTINATION(last_destination | bin);
```

The final step is the trivial unfold of the atomic statement ▸$_3$ using the UnfoldAtomic method. At this point the user marks the *OptimizePEAS* goal as achieved.

C.4. Map DID_NOT_SET_SWITCH_WHEN_HAD_CHANCE

In this section, we will assume the user has turned his attention to mapping away the global constraints in the *spec*. In our portion of the router spec, there is only one: DID_NOT_SET_SWITCH_WHEN_HAD_CHANCE.

```
constraint DID_NOT_SET_SWITCH_WHEN_HAD_CHANCE
  always prohibit ∃ package,switch ||
   (package:LOCATED_AT = switch
     and
    SWITCH_SET_WRONG_FOR_PACKAGE(switch,package)
     and
    ((package = first(PACKAGES_DUE_AT_SWITCH(*,switch))
       and
     SWITCH_IS_EMPTY(switch)) asof everbefore));
```

STEP 4.1 *(user)*: *Map* DID_NOT_SET_SWITCH_WHEN_HAD_CHANCE

| **Method** MapConstraintAsDemon |

Goal: *Map* C|*constraint*

Action: 1) *Reformulate* C as always prohibit P

2) *Show* IMPLIED_BY(Q, P)

3) *Apply* REFORMULATE_CONSTRAINT_AS_DEMON(C, Q, D_{new})

4) *Map* D_{new}

[To map a prohibitive constraint, first choose some predicate Q that is always true when the constraint is violated, and then introduce a demon whose trigger is Q and whose body is a requirement of ~P.]

| **End Method** |

STEP 4.2: *Show*

```
 ∃ package,switch ||
▶1  (package:LOCATED_AT = switch
      and
▶2   SWITCH_SET_WRONG_FOR_PACKAGE(switch,package)
      and
▶3   ((package = first(PACKAGES_DUE_AT_SWITCH(*,switch))
        and
      SWITCH_IS_EMPTY(switch)) asof everbefore));

implies Q
```

```
| Method  ConjunctImpliesConjunctArm                              |

        Goal: Show X|conjunction implies Y
        Filter: a) unbound[Y]
                b) conjuct-arm[A|logical-expression, X]
        Action: 1) Assert X implies A

        [(P1 and P2 and ...Pn) implies Pi]
| End Method                                                      |
```

There are three possible choices for A corresponding to the three conjunct arms:

1. ▶$_1$ Trigger when a package becomes located at a switch; guarantee that either the switch is set right or that there never was a chance to set it right[58].

2. ▶$_2$ Trigger when the switch is set wrong; guarantee that the package is not at the switch or that there never was a chance to set the switch right.

3. ▶$_3$ Trigger when there is a chance to set the switch right; guarantee that the package is not at the switch or that the switch is set right.

We will choose the third:

```
((package = first(PACKAGES_DUE_AT_SWITCH(*,switch))
   and
 SWITCH_IS_EMPTY(switch)) asof everbefore)
```

The effect of REFORMULATE_CONSTRAINT_AS_DEMON can be characterized as follows:

[58] Actually, you only have to make this guarantee as long as the triggering predicate holds. This is true for the other two cases as well.

```
    always prohibit P
=>
    demon
     trigger Q
     response require (~P from ThisEvent until ~Q)

  where P implies Q
```

Define a demon who triggers on Q and posts a requirement that P not be true between the time the demon triggers (Q becomes true) and Q becomes false.

After application of this transformation (and a straightforward removal of the historical reference from the trigger and simplification of the requirement conjunction), we have the following:

```
...
demon SET_SWITCH_WHEN_HAVE_CHANCE(switch, package)
  trigger (package = first(PACKAGES_DUE_AT_SWITCH(*,switch))
              and
           SWITCH_IS_EMPTY(switch))
  response
    require (~(package:LOCATED_AT = switch
                  and
             SWITCH_SET_WRONG_FOR_PACKAGE(switch,package))
                from ThisEvent[59]
▸2               until ~((package =
                           first(PACKAGES_DUE_AT_SWITCH(*,switch))
                          and
                         SWITCH_IS_EMPTY(switch)) asof everbefore))
```

The response of the new demon should be read as "require that the package not be located at the switch when the switch is set wrong. Make sure that this is true from the time the demon triggers *until* the switch is not ready to be set, >> asof everbefore <<". The until clause is clearly false since the trigger implies that the switch has been ready to be set in the past. A simple transformation of the until clause ▸2,

```
    ... until false    =>    until evermore
```

allows us to simplify (SET_SWITCH ▸1 is included for context):

---

[59] i.e., the triggering of this demon.

```
▸1 demon SET_SWITCH(switch)
    trigger RANDOM()
    response
      begin
        require SWITCH_IS_EMPTY(switch);
        update :SWITCH_SETTING of switch to switch:SWITCH_OUTLET
      end;

demon SET_SWITCH_WHEN_HAVE_CHANCE(switch, package)
  trigger (package = first(PACKAGES_DUE_AT_SWITCH(*,switch))
                and
              SWITCH_IS_EMPTY(switch))
  response
    require (~(package:LOCATED_AT = switch
                    and
               SWITCH_SET_WRONG_FOR_PACKAGE(switch,package))
                  from ThisEvent
▸2                until evermore
```

STEP 4.3: *Map* SET_SWITCH_WHEN_HAVE_CHANCE

```
| Method  MapByConsolidation                                    |

        Goal:   Map D|demon
        Filter: a) pattern-match[demon, D2, spec]
                b) D ≠ D2
        Action: 1) Consolidate D and D2

        [To map D, find some other demon D2 and consolidate.]
| End Method                                                    |
```

A separate method will be triggered for each binding of D2, one for each demon in the program. We will choose the binding to SET_SWITCH.

STEP 4.4: *Consolidate* SET_SWITCH with SET_SWITCH_WHEN_HAVE_CHANCE

| Method MergeDemons |

Goal: *Consolidate* D1|*demon* and D2|*demon*

Action: 1) *Equivalence* trigger-of[D1] and trigger-of[D2]

2) *Equivalence* var-declaration-of[D1] and var-declaration-of[D2]

3) *Show* MERGEABLE_DEMONS(D1, D2, I|*ordering*)

4) *Apply* DEMON_MERGE(D1, D2, I)

[You can consolidate two demons if you can show that they have the same local variables, the same triggering pattern and that they meet certain merging conditions.]

| End Method |

STEP 4.5: *Equivalence*

```
  trigger RANDOM()
and
  trigger package = first(PACKAGES_DUE_AT_SWITCH(*,switch))
            and
        SWITCH_IS_EMPTY(switch)
```

| Method Anchor2 |

Goal: *Equivalence* X *and* Y

Action: 1) *Reformulate* X *as* Y

[Try changing the first construct into something that matches the second.]

| End Method |

STEP 4.6: *Reformulate* RANDOM() as

```
package = first(PACKAGES_DUE_AT_SWITCH(*,switch))
    and
SWITCH_IS_EMPTY(switch)
```

```
| Method  SpecializeRandom                                              |

        Goal: Reformulate X|RANDOM as Y|expression
        Action: 1) Show NON_EMPTY_SPECIALIZATION(Y)
                2) Apply
                        REPLACE_RANDOM_WITH_SPECIALIZATION(X Y)

        [You can always replace RANDOM with a more specialized event if you can
        show the new event does not remove all choices.]
| End Method                                                            |
```

We rely on the user to show that a non-empty subset of triggerings remain for SET_SWITCH.

After the application of REPLACE_RANDOM_WITH_SPECIALIZATION, we have

```
...
demon SET_SWITCH(switch, package)
  trigger package = first(PACKAGES_DUE_AT_SWITCH(*,switch))
                  and
            SWITCH_IS_EMPTY(switch)
  response
    begin
      update :SWITCH_SETTING of switch to switch:SWITCH_OUTLET
                 where SWITCH_IS_EMPTY(switch)
    end;

demon SET_SWITCH_WHEN_HAVE_CHANCE(switch, package)
  trigger (package = first(PACKAGES_DUE_AT_SWITCH(*,switch))
                  and
              SWITCH_IS_EMPTY(switch))
  response
    require (~(package:LOCATED_AT = switch
                   and
               SWITCH_SET_WRONG_FOR_PACKAGE(switch,package))
                       from ThisEvent
                       until evermore
```

Our Equivalence goal has been achieved and we can consolidate the two demons.

```
...
demon SET_SWITCH(switch, package)
  trigger package = first(PACKAGES_DUE_AT_SWITCH(*,switch))
                  and
             SWITCH_IS_EMPTY(switch)

  response
    begin
      update :SWITCH_SETTING of switch to switch:SWITCH_OUTLET
          where SWITCH_IS_EMPTY(switch);
▸1     require (~(package:LOCATED_AT = switch
                    and
                 SWITCH_SET_WRONG_FOR_PACKAGE(switch,package))
                   from ThisEvent
                   until evermore
    end;
```

We have removed the global constraint DID_NOT_SET_SWITCH_WHEN_HAD_CHANCE from the program, but are left with a residual local constraint ▸$_1$ within SET_SWITCH.

STEP 4.7 *(user)*: *Map*

```
▸1     require (~(package:LOCATED_AT = switch
                    and
                   SWITCH_SET_WRONG_FOR_PACKAGE(switch,package))
                     from ThisEvent
                     until evermore
```

| Method CasifyPosConstraint |

Goal: *Map* C | +*constraint*

Action: 1) *Casify* C

2) forall case-of[X, C] do *Map* X

[Try mapping by case analysis.]

| End Method |

The remainder of the development in this section will be based on a number of different case analysis strategies for removing the requirements in the SET_SWITCH demon. The interaction between the user and system during this time points out the fundamental role of

each: the system suggests rather broad strategies with keystone pieces left unbound; the user selects among the strategies based on his ability to fill in the missing pieces. The latter activity requires what we might call the insightful or intelligent component of reasoning; we suspect that such activity will resist automation for some time to come.

STEP 4.8: *Casify*

```
▶1     require (~(package:LOCATED_AT = switch
              and
            SWITCH_SET_WRONG_FOR_PACKAGE(switch,package))
             from ThisEvent
             until evermore
```

```
| Method  CasifyFromUntilEverConstraint                         |

      Goal: Casify C | +constraint
      Action: 1) Reformulate C as
                                  P from E until evermore
              2) Apply CASIFY_AS_NOW_AND_AFTER(C)

      [You can show that C holds from E until everafter if you can show it holds at E
      and after E.]
| End Method                                                    |
```

This method makes the following transformation

```
     +constraint P from E until evermore
 ⇒
     +constraint P at E;
     +constraint P after E;
```

In our case, this means showing that either the package is not located at the switch or that the switch is set right at the time the demon triggered ▶$_1$ and for all time after ▶$_2$. After application of CASIFY_AS_NOW_AND_AFTER, we have[60]

[60]Note that the reformulation goal is trivially satisfied. This is because earlier we carried out the reformulation for clarity. Normally this would be carried out here where it is well motivated.

```
...
demon SET_SWITCH(switch, package)
  trigger package = first(PACKAGES_DUE_AT_SWITCH(*,switch))
                  and
                SWITCH_IS_EMPTY(switch)

  response
    begin
      update :SWITCH_SETTING of switch to switch:SWITCH_OUTLET
          where SWITCH_IS_EMPTY(switch);
▸1      require (~(package:LOCATED_AT = switch
                      and
                 SWITCH_SET_WRONG_FOR_PACKAGE(switch,package))
                   at ThisEvent;
▸2      require (~(package:LOCATED_AT = switch
                      and
                 SWITCH_SET_WRONG_FOR_PACKAGE(switch,package))
                   after ThisEvent

    end;
```

STEP 4.9: *Map*

```
▸1    require (~(package:LOCATED_AT = switch
                     and
               SWITCH_SET_WRONG_FOR_PACKAGE(switch,package))
                 at ThisEvent
```

```
| Method  TriggerImpliesConstraint                                  |

        Goal: Map  R|require
        Filter: a) component-of[R, D|demon]
        Action: 1) Reformulate R as require P at ThisEvent
                2) Show IMPLIED_BY(P, trigger-of[D])
                3) Apply REMOVE_IMPLIED_REQUIREMENT(R)

        [If a requirement is part of a demon, try showing that it is implied by the
        demon's trigger.]
| End Method                                                        |
```

We rely on the user to verify that the trigger does indeed imply the constraint, i.e., a switch being empty implies that the package is not located there. This removes the first case. We now must tackle the more interesting second case.

STEP 4.10: *Map*

```
▸2   require (~(package:LOCATED_AT = switch
                 and
               SWITCH_SET_WRONG_FOR_PACKAGE(switch,package))
                after ThisEvent
```

| Method CasifyPosConstraint |

Goal: *Map* C | +*constraint*

Action: 1) *Casify* C

2) forall case-of[X, C] do *Map* X

[Try mapping by case analysis.]

| End Method |

STEP 4.11: *Casify*

```
▸2   require (~(package:LOCATED_AT = switch
                 and
               SWITCH_SET_WRONG_FOR_PACKAGE(switch,package))
                after ThisEvent
```

| Method CasifyAroundEvent |

Goal: *Casify* C | *constraint*

Action: 1) *Reformulate* C as *constraint* P after E

2) *Show* FUTURE_EVENT(F, E)

3) *Apply* CASIFY_AROUND_EVENT(C, F)

[Choose some event F in the future and show that C holds before, during and after F.]

| End Method |

This method splits a constraint into three cases: 1) before some future event F, 2) during F and 3) after F. In this case, the difficult task is picking the right future event F. We rely on the user to make this choice:

bind F to *package*:LOCATED_AT = *switch*

After application of CASIFY_AROUND_EVENT, we have our before ▸$_1$, during ▸$_2$ and after ▸$_3$ cases:

```
...
demon SET_SWITCH(switch, package)
  trigger package = first(PACKAGES_DUE_AT_SWITCH(*,switch))
                and
             SWITCH_IS_EMPTY(switch)

  response
    begin
▶0     update :SWITCH_SETTING of switch to switch:SWITCH_OUTLET
          where SWITCH_IS_EMPTY(switch);
▶1     require (~(package:LOCATED_AT = switch
                    and
                 SWITCH_SET_WRONG_FOR_PACKAGE(switch,package))
                   after ThisEvent until package:LOCATED_AT = switch;
▶2     require (~(package:LOCATED_AT = switch
                    and
                 SWITCH_SET_WRONG_FOR_PACKAGE(switch,package))
                   during package:LOCATED_AT = switch;
▶3     require (~(package:LOCATED_AT = switch
                    and
                 SWITCH_SET_WRONG_FOR_PACKAGE(switch,package))
                   after package:LOCATED_AT = switch;

    end;
```

Again, we must map each of the new cases.

STEP 4.12: *Map*

```
▶1    require (~(package:LOCATED_AT = switch
                   and
                SWITCH_SET_WRONG_FOR_PACKAGE(switch,package))
                  after ThisEvent until package:LOCATED_AT = switch;
```

```
| Method  NotXUntilX                                          |

      Goal: Map  R| +constraint
      Action: 1) Reformulate R as +constraint P until E
              2) Show IMPLIED_BY(P, ~E)
              3) Apply REMOVE_VACUOUS_CONSTRAINT(R)

      [P until E ⇒ true when ~E implies P]
| End Method                                                  |
```

We rely on the user to show that the negation of the until clause -- the package is not located at the switch -- implies the predicate. We can thus remove the first requirement ▶$_1$. By (the user) showing that the package will never again return to the switch after it leaves it, we can similarly remove the third requirement ▶$_3$. This leaves us with the second requirement ▶$_2$.

STEP 4.13: *Map*

```
▶2    require (~(package:LOCATED_AT = switch
                and
              SWITCH_SET_WRONG_FOR_PACKAGE(switch,package))
                during package:LOCATED_AT = switch;
```

We can simplify this to

```
    require ~SWITCH_SET_WRONG_FOR_PACKAGE(switch,package)
                during package:LOCATED_AT = switch;
```

We will again use case analysis to simplify the problem.

```
| Method  CasifyPosConstraint                          |

      Goal:  Map C|+constraint
      Action: 1) Casify C
              2) forall case-of[X, C] do Map X

      [Try mapping by case analysis.]
| End Method                                           |
```

STEP 4.14: *Casify*

```
    require ~SWITCH_SET_WRONG_FOR_PACKAGE(switch,package)
                during package:LOCATED_AT = switch;
```

```
| Method  PastInduction                                              |

     Goal: Casify  C | +constraint
     Action: 1) Reformulate C as +constraint P during E
             2) Show EVENT_BEFORE_EVENT(B, E)
             3) Apply PAST_INDUCTION_CASIFY(C, B)

     [Use induction from some past state.]
| End Method                                                         |
```

This method makes the following transformation:

```
     +constraint P during E
  ⟹
     +constraint P at B || B before E
     +constraint ~(start of ~P) between B, after E
```

To paraphrase, there exists some state B before E where P holds and P does not change between B and E. The choice of B is naturally critical and is left to the user:

```
        bind B to last update of switch:SWITCH_SETTING in SET_SWITCH (▸0)
```

After application of PAST_INDUCTION_CASIFY, we have

```
...
demon SET_SWITCH(switch, package)
  trigger package = first(PACKAGES_DUE_AT_SWITCH(*,switch))
                  and
                SWITCH_IS_EMPTY(switch)
  response
    begin
▸0     update :SWITCH_SETTING of switch to switch:SWITCH_OUTLET
          where SWITCH_IS_EMPTY(switch);
▸1     require  ~SWITCH_SET_WRONG_FOR_PACKAGE(switch,package)
              at last update of switch:SWITCH_SETTING;
▸2     require
            ~(start of ~SWITCH_SET_WRONG_FOR_PACKAGE(switch,package))
              between last update of switch:SWITCH_SETTING,
                                         package:LOCATED_AT = switch
    end;
```

STEP 4.15: *Map*

```
▸1      require ~SWITCH_SET_WRONG_FOR_PACKAGE(switch,package)
             at last update of switch:SWITCH_SETTING;
```

```
| Method  MoveConstraintToAction                                  |

        Goal:  Map C|require
        Action: 1) Reformulate C as
                                  require P at last E|Action-event
                2) Show LAST_ACTION(A|action, E)
                3) Apply MOVE_CONSTRAINT_TO_ACTION(C, A)

        [If a constraint C is on some action event E at A, attach the constraint to A.]
| End Method                                                      |
```

We rely on the user to show that the update of the switch setting ▸1 in SET_SWITCH is the only update of a switch setting and hence, it must have been the last. After application of MOVE_CONSTRAINT_TO_ACTION, we have

```
...
demon SET_SWITCH(switch, package)
  trigger package = first(PACKAGES_DUE_AT_SWITCH(*,switch))
                 and
             SWITCH_IS_EMPTY(switch)
  response
    begin
▸0    update :SWITCH_SETTING of switch to switch:SWITCH_OUTLET
          where SWITCH_IS_EMPTY(switch)
                 and
               ~SWITCH_SET_WRONG_FOR_PACKAGE(switch,package);
▸2    require
          ~(start of ~SWITCH_SET_WRONG_FOR_PACKAGE(switch,package))
            between last update of switch:SWITCH_SETTING,
                                  package:LOCATED_AT = switch
    end;
```

STEP 4.16: *Map*

```
▸2      require
            ~(start of ~SWITCH_SET_WRONG_FOR_PACKAGE(switch,package))
              between last update of switch:SWITCH_SETTING,
                                             package:LOCATED_AT = switch
```

```
| Method   ShowNoChange                                         |

        Goal: Map C| +constraint ~(start of P)
                           between E1,E2
        Action: 1) Show UNCHANGED_BETWEEN_EVENTS(P, E1, E2)
                2) Apply  REMOVE_UNCHANGED_CONSTRAINT(C)

        [The direct approach.]
| End Method                                                    |
```

STEP 4.17: *Show*

~(start of ~SWITCH_SET_WRONG_FOR_PACKAGE(*switch,package*))
between last update of *switch*:SWITCH_SETTING, *package*:LOCATED_AT = *switch*

Showing that the switch is never set wrong (relative to a particular package) once it is set right lies beyond the capabilities of the system. We rely on tne user to assert the necessary property.

After application of REMOVE_UNCHANGED_CONSTRAINT, we have

```
...
demon SET_SWITCH(switch, package)
  trigger package = first(PACKAGES_DUE_AT_SWITCH(*,switch))
                and
              SWITCH_IS_EMPTY(switch)
  response
▸0    update :SWITCH_SETTING of switch to switch:SWITCH_OUTLET
          where SWITCH_IS_EMPTY(switch)
                    and
                  ~SWITCH_SET_WRONG_FOR_PACKAGE(switch,package);
```

Our last task will be to map the non-deterministic choice of switch settings ▸$_0$ using the attached constraints as a guide.

STEP 4.18 *(user)*: *Map*

```
▸0      update :SWITCH_SETTING of switch to switch:SWITCH_OUTLET
          where SWITCH_IS_EMPTY(switch)
                    and
                 ~SWITCH_SET_WRONG_FOR_PACKAGE(switch,package);
```

```
| Method   ComputeNewValue                                        |

      Goal: Map U|update X of Y to Z where P
      Action: 1) Apply
            COMPUTE_DERIVED_OBJECT_FROM_CONSTRAINT(U)

      [Reformulate Z as derived object using P.]
| End Method                                                       |
```

The application of COMPUTE_DERIVED_OBJECT_FROM_CONSTRAINT gives us

```
...
demon SET_SWITCH(switch, package)
  trigger package = first(PACKAGES_DUE_AT_SWITCH(*,switch))
                  and
                SWITCH_IS_EMPTY(switch)
  response
    update :SWITCH_SETTING of switch to
               (pipe || pipe =  switch:SWITCH_OUTLET
                        and
                     SWITCH_IS_EMPTY(switch)
                        and
▸1                   ~SWITCH_SET_WRONG_FOR_PACKAGE(switch,package);
```

STEP 4.19 *(user)*: *Unfold* SWITCH←SET←WRONG←FOR←PACKAGE at ▸1

| **Method** ScatterComputationOfDerivedRelation |

Goal: *Unfold* DR|***derived-relation*** at L
Filter: a) reference-location[DR, L, $]
Action: 1) *Apply* UNFOLD_COMPUTATION_CODE(DR L)
2) *Purify* L

[To unfold a derived relation DR at a reference point, stick in code to compute it and make sure L is within implementable portion of spec.]

| **End Method** |

Unfolding SWITCH_SET_WRONG_FOR_PACKAGE ▶$_1$ and simplifying (see example A, section E.14) gives us

---

```
...
demon SET_SWITCH(switch, package)
  trigger package = first(PACKAGES_DUE_AT_SWITCH(*,switch))
                  and
                 SWITCH_IS_EMPTY(switch)
  response
     update :SWITCH_SETTING of switch to
             (pipe || pipe =  switch:SWITCH_OUTLET
                      and
▶2                SWITCH_IS_EMPTY(switch)
                      and
                  LOCATION_ON_ROUTE_TO_BIN(pipe,
                                  package:DESTINATION));
```

Finally, we can get rid of the empty switch constraint ▶$_2$ under our assumption that the response of a demon is executed in the same state as it was triggered:

```
...
demon SET_SWITCH(switch, package)
  trigger package = first(PACKAGES_DUE_AT_SWITCH(*,switch))
               and
              SWITCH_IS_EMPTY(switch)
  response
     update :SWITCH_SETTING of switch to
               (pipe || pipe =  switch:SWITCH_OUTLET
                         and
                  LOCATION_ON_ROUTE_TO_BIN(pipe,
                                             package:DESTINATION));
```

C.5. Map PACKAGES_DUE_AT_SWITCH

We will focus our attention on the derived relation PACKAGES_DUE_AT_SWITCH:

```
relation PACKAGES_DUE_AT_SWITCH(packages_due | sequence of package,
                                                              switch)
    definition packages_due =
       {a package ||
             LOCATION_ON_ROUTE_TO_BIN(switch package:DESTINATION)
                and
             ~((package:LOCATED_AT = switch) asof everbefore)
                and
             ~MISROUTED(package)
       } ordered temporally by start (package:LOCATED_AT = the source));
```

Abstractly, the sequence of packages is defined in terms of

{S} <u>ordered</u> <u>with</u> <u>respect</u> <u>to</u> Event

A *package* is in the set of packages S if conjunctively

- LOCATION_ON_ROUTE_TO_BIN(*switch*, *package*:DESTINATION) i.e., the *switch* lies on route to the *package*'s destination.
- ~((*package*:LOCATED_AT = *switch*) <u>asof</u> <u>everbefore</u>), i.e., the *package* has not already reached the *switch*.
- ~MISROUTED(*package*), i.e., the *package* is still expected to show up at some future time at the *switch*.

<u>STEP 5.1</u> *(user)*: *Map* PACKAGES_DUE_AT_SWITCH

As in previous sections, we have two basic strategic choices: compute on demand; compute on change. We will choose the latter here.

```
| Method  MaintainDerivedRelation                                    |

        Goal: Map DR|derived-relation
        Action: 1) MaintainIncrementally DR

        [One way of mapping a derived relation is to maintain it explicitly.]
| End Method                                                         |
```

STEP 5.2: *MaintainIncrementally* PACKAGES_DUE_AT_SWITCH

```
| Method  ScatterMaintenanceForDerivedRelation                       |

        Goal: MaintainIncrementally DR
        Filter: a) gist-type-of[DR, derived-relation]
        Action: 1) Flatten body-of[DR]
                2) forall reference-location[BR, $, DR]
                 do forall reference-location[BR, L, spec)
                       do begin
                         Apply INTRODUCE_MAINTENANCE_CODE(DR L)
                         Purify L
                        end

        [To maintain a derived relation DR, find everywhere the base relations of DR
        are changed and stick code in to maintain. Make sure that all base relations
        are simple before maintenance and that all code is pure after.]
| End Method                                                         |
```

STEP 5.3: *Flatten* PACKAGES_DUE_AT_SWITCH

```
| Method  Flatten                                                    |

        Goal: Flatten DR|derived-relation
        Action: 1) forall
            reference-location[BR|derived-relation,$,DR]
                          do Map BR

        [Map all derived relations found in DR into simple ones.]
| End Method                                                         |
```

Before maintaining, we must first get rid of any nested derived relations. There are currently two: LOCATION_ON_ROUTE_TO_BIN and MISROUTED.

STEP 5.4: *Map* LOCATION_ON_ROUTE_TO_BIN

```
relation LOCATION_ON_ROUTE_TO_BIN(LOCATION,BIN)
   definition
      case LOCATION of
        BIN   ⇒ LOCATION = BIN;
        PIPE
          ⇒   LOCATION_ON_ROUTE_TO_BIN(
                 LOCATION:connection_to_switch_or_bin,BIN);
        SWITCH
          ⇒ LOCATION_ON_ROUTE_TO_BIN(LOCATION:switch_outlet,BIN);
        SOURCE
          ⇒ LOCATION_ON_ROUTE_TO_BIN(LOCATION:source_outlet,BIN);
      end case;
```

We can either choose to compute LOCATION←ON←ROUTE←TO←BIN on demand (i.e., unfolding it) or maintain it explicitly. Since the relation is static, maintenance looks most promising.

```
| Method  StoreExplicitly                                          |

        Goal:  Map DR | derived-relation
        Filter:  a) STATIC(DR)
        Action:  1) Show FINITE_EXPLICATION(DR)
                 2) Apply INITIALIZE_MEMO_RELATION(M, DR)
                 3) forall location-reference[DR, L, spec]
                            do Apply REPLACE-REF-WITH-MEMO(L, M)
                 4) Apply REMOVE_UNREFERENCED_RELATION(DR)

        [You can explicitly compute a static derived relation given a finite number of
        resulting db insertions.]
| End Method                                                       |
```

INITIALIZE_MEMO_RELATION will define a new memo relation and code to initialize it.

```
...
relation MEMO_LOCATION_BIN(location, bin);

demon INITIALIZE_MEMO_LOCATION_BIN()
   trigger: (start initialization_state)61
   response
      loop L | LOCATION do
         loop B | BIN || LOCATION_ON_ROUTE_TO_BIN(L, B) do
            insert MEMO_LOCATION_BIN(L, B);
...
```

We can now replace references to LOCATION_ON_ROUTE_TO_BIN with corresponding references to MEMO_LOCATION_BIN trivially except for the initialization above. Here, we will use some loop transformations to get

```
...
relation MEMO_LOCATION_BIN(location, bin);

demon INITIALIZE_MEMO_LOCATION_BIN()
   trigger: (start initialization_state)
   response
    begin
      loop B | BIN do insert MEMO_LOCATION_BIN(B, B);
      loop L | LOCATION ||
                        MEMO_LOCATION_BIN(L, B) and
                        L = L2:CONNECTION_TO_SWITCH_OR_BIN
             do insert MEMO_LOCATION_BIN(L2, B);
    end
...
```

We next have to deal with the derived-relation MISROUTED.

STEP 5.5: *Map* MISROUTED

[61] A special state proceeding the start-up of a system.

```
relation MISROUTED(package)
    definition
        ~MEMO_LOCATION_BIN(package:LOCATED_AT, package:DESTINATION)
            or
        SWITCH_SET_WRONG_FOR_PACKAGE(package:c(located__at),
                                                   package);
```

To paraphrase, a *package* is misrouted if either its current location is not on the route to its destination or if it is at a switch, the switch is set wrong.

In the case of this derived relation, we will try a backward inference strategy of computing the relation on demand.

```
| Method  UnfoldDerivedRelation                                  |

        Goal: Map DR|derived-relation
        Action: 1) forall reference-location[DR, L, spec]
                          do Unfold DR at L

        [One way of eliminating a derived relation is to unfold it at its reference
        points.]
| End Method                                                     |
```

STEP 5.6: *Unfold* MISROUTED at PACKAGES_DUE_AT_SWITCH

```
| Method  ScatterComputationOfDerivedRelation                    |

        Goal: Unfold DR|derived-relation at L
        Filter: a) reference-location[DR, L, $]
        Action: 1) Apply UNFOLD_COMPUTATION_CODE(DR L)
                2) Purify L

        [To unfold a derived relation DR at a reference point, stick in code to compute
        it and make sure L is within implementable portion of spec.]
| End Method                                                     |
```

```
relation PACKAGES_DUE_AT_SWITCH(packages_due | sequence of package,
                                                          switch)
    definition packages_due =
       {a package ||
            MEMO_LOCATION_BIN(switch package:DESTINATION)
              and
            ~((package:LOCATED_AT = switch) asof everbefore)
              and
            ~(~MEMO_LOCATION_BIN(package:LOCATED_AT,
                                           package:DESTINATION)
                  or
                SWITCH_SET_WRONG_FOR_PACKAGE(package:LOCATED_AT,
                                                  package))
       } ordered temporally by start (package:LOCATED_AT = the source));
```

The Flatten method has completed, but a new derived-relation has been introduced: SWITCH_SET_WRONG_FOR_PACKAGE, i.e., the Flatten goal has not been achieved. The goal will be re-activated.

STEP 5.7: *Flatten* PACKAGES_DUE_AT_SWITCH

```
| Method  Flatten                                          |

        Goal:  Flatten DR|derived-relation
        Action:  1) forall
            reference-location[BR|derived-relation,$,DR]
                              do Map BR

        [Map all derived relations found in DR into simple ones.]
| End Method                                               |
```

PACKAGES_DUE_AT_SWITCH now relies upon the derived relation SWITCH_SET_WRONG_FOR_PACKAGE which was introduced in the unfolding of MISROUTED.

relation SWITCH_SET_WRONG_FOR_PACKAGE(*switch, package*)
  definition
    MEMO_LOCATION_BIN(*switch*, *package*:DESTINATION)
      and
    ~MEMO_LOCATION_BIN(*switch*:SWITCH_SETTING, *package*:DESTINATION)

To paraphrase, a switch is set wrong for a package if the switch is along the route to the package's destination and its current setting is not.

STEP 5.8: *Map* SWITCH_SET_WRONG_FOR_PACKAGE

| Method UnfoldDerivedRelation |

Goal: *Map* DR|*derived-relation*

Action: 1) forall reference-location[DR, L, *spec*]
do *Unfold* DR at L

[One way of eliminating a derived relation is to unfold it at its reference points.]

| End Method |

STEP 5.9: *Unfold* SWITCH_SET_WRONG_FOR_PACKAGE at PACKAGES_DUE_AT_SWITCH

| Method ScatterComputationOfDerivedRelation |

Goal: *Unfold* DR|*derived-relation* at L

Filter: a) reference-location[DR, L, $]

Action: 1) *Apply* UNFOLD_COMPUTATION_CODE(DR L)
2) *Purify* L

[To unfold a derived relation DR at a reference point, stick in code to compute it and make sure L is within implementable portion of spec.]

| End Method |

Unfolding SWITCH_SET_WRONG_FOR_PACKAGE in PACKAGES_DUE_AT_SWITCH we have

```
relation PACKAGES_DUE_AT_SWITCH(packages_due | sequence of package,
                                                          switch)
    definition packages_due =
       {a package ||
              MEMO_LOCATION_BIN(switch package:DESTINATION)
                 and
              ~((package:LOCATED_AT = switch) asof everbefore)
                 and
▸1            ~(~MEMO_LOCATION_BIN(package:LOCATED_AT,
                                                   package:DESTINATION)
                        or
                 ∃ switch.2 ||
                    (package:LOCATED_AT = switch.2
                        and
                     MEMO_LOCATION_BIN(switch.2, package:DESTINATION)
                        and
                     ~MEMO_LOCATION_BIN(switch.2:SWITCH_SETTING,
                                                    package:DESTINATION)))
       } ordered temporally by start (package:LOCATED_AT = the source));
```

Distributing the negation through the third term (▸$_1$) gives us

```
relation PACKAGES_DUE_AT_SWITCH(packages_due | sequence of package,
                                                          switch)
    definition packages_due =
       {a package ||
              MEMO_LOCATION_BIN(switch package:DESTINATION)
                 and
              ~((package:LOCATED_AT = switch) asof everbefore)
                 and
▸2              (MEMO_LOCATION_BIN(package:LOCATED_AT,
                                                    package:DESTINATION)
                   and
                 ~∃ switch.2 ||
                    (package:LOCATED_AT = switch.2
                        and
▸3                   MEMO_LOCATION_BIN(switch.2, package:DESTINATION)
                        and
                    ~MEMO_LOCATION_BIN(switch.2:SWITCH_SETTING,
                                                    package:DESTINATION)))
       } ordered temporally by start (package:LOCATED_AT = the source));
```

Finally, we can show that the third term ▸$_2$ implies that our current location is on route to our destination (▸$_3$) and therefore that if we are at a switch, it is on route to our destination:

```
relation PACKAGES_DUE_AT_SWITCH(packages_due | sequence of package,
                                                    switch)
   definition packages_due =
      {a package ||
            MEMO_LOCATION_BIN(switch package:DESTINATION)
               and
            ~((package:LOCATED_AT = switch) asof everbefore)
               and
            (MEMO_LOCATION_BIN(package:LOCATED_AT,
                                              package:DESTINATION)
                     and
             ~∃ switch.2 ||
                  (package:LOCATED_AT = switch.2
                     and
                   ~MEMO_LOCATION_BIN(switch.2:SWITCH_SETTING,
                                              package:DESTINATION)))
      } ordered temporally by start (package:LOCATED_AT = the source));
```

We have now flattened the body of PACKAGES_DUE_AT_SWITCH and are ready to scatter the maintenance code. The locations of interest are

1. where *package*:DESTINATION changes · CREATE_PACKAGE

2. where *package*:LOCATION changes, i.e., negates the second term · CREATE_PACKAGE, RELEASE_PACKAGE_INTO_NETWORK, MOVE_PACKAGE

3. where :SWITCH_SETTING changes · SET_SWITCH

The high level view of the incremental maintenance process we will use is as follows: 1) when a package enters the network, for each switch S that is on the route to the package's destination bin, *append* the package to the sequence of package's due at S, 2) when the right conditions occur -- the package enters S or becomes misrouted before reaching S -- remove the package from S's sequence.

Looking first at CREATE_PACKAGE, we loop ▸$_1$ through the free variable *switch* and add ▸$_2$ the newly created *package.new* to the sequence for all switches meeting the criteria.

```
demon CREATE_PACKAGE()
    trigger RANDOM()
    response
      atomic
        create package.new ||
            package.new:DESTINATION = a bin and
            package.new:LOCATED_AT = the source;
▸1      loop switch ||
            MEMO_LOCATION_BIN(switch package.new:DESTINATION)
               and
            ~((package.new:LOCATED_AT = switch) asof everbefore)
               and
            (MEMO_LOCATION_BIN(package.new:LOCATED_AT,
                              package.new:DESTINATION)
               and
             ~∃ switch.2 ||
                 (package.new:LOCATED_AT = switch.2
                    and
                  ~MEMO_LOCATION_BIN(switch.2:SWITCH_SETTING,
                                    package.new:DESTINATION)))
▸2       do update packages_due of PACKAGES_DUE_AT_SWITCH(switch,$)
           to PACKAGES_DUE_AT_SWITCH(switch,*) concat <package.new>
      end atomic;
```

Reasoning that *package.new* cannot have been at (any) *switch*, that it certainly must be on the route to its bin (unless a pipe is missing) and that it is not currently located at a switch allows us to simplify to the following:

```
demon CREATE_PACKAGE()
    trigger RANDOM()
    response
      atomic
        create package.new ||
            package.new:DESTINATION = a bin and
            package.new:LOCATED_AT = the source;
▸3       loop (switch ||
          MEMO_LOCATION_BIN(switch, package.new:DESTINATION))
          do update packages_due of PACKAGES_DUE_AT_SWITCH(switch,$)
            to PACKAGES_DUE_AT_SWITCH(switch,*) concat <package.new>
      end atomic;
```

CREATE_PACKAGE is outside of our portion of the development, hence the introduced code ▸$_3$ must be moved in.

STEP 5.10: *Purify* loop ... do ... in CREATE_PACKAGE

```
| Method  PurifyDemon                                          |

        Goal: Purify A|action in D|demon
        Action: 1) Remove L from D

        [Remove unpure statement L from D.]
| End Method                                                   |
```

STEP 5.11: *Remove*

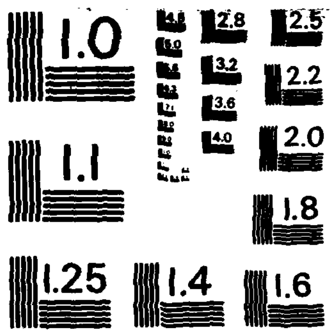
```
▸3      loop (switch || MEMO_LOCATION_BIN(switch,
                                  package.new:DESTINATION))
      do update packages_due of PACKAGES_DUE_AT_SWITCH(switch,$)
       to PACKAGES_DUE_AT_SWITCH(switch,*) concat <package.new>;

      from CREATE_PACKAGE
```

```
| Method  RemoveFromDemon                                      |

        Goal: Remove A|action from D|demon
        Action: 1) Globalize A
                2) forall trigger-location[D2|demon, body-of[*, D], spec]
                       do Apply MOVE_STATEMENT_TO_DEMON(A, D2)

        [Find all demons that trigger from D and move the action A there.]
| End Method                                                   |
```

STEP 5.12: *Globalize*

```
      loop (switch || MEMO_LOCATION_BIN(switch,
                                  package.new:DESTINATION))
      do update packages_due of PACKAGES_DUE_AT_SWITCH(switch,$)
       to PACKAGES_DUE_AT_SWITCH(switch,*) concat <package.new>;
```

AD-A139 918 AUTOMATING THE TRANSFORMATIONAL DEVELOPMENT OF SOFTWARE 2/3
VOLUME 2 APPENDICES(U) UNIVERSITY OF SOUTHERN
CALIFORNIA MARINA DEL REY INFORMATION S.. S F FICKAS
UNCLASSIFIED MAR 83 ISI/RR-83-109 NSF-MCS79-18792 F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

| **Method** GlobalizeAction |

Goal: *Globalize* A|*action*
Filter: a) component-of[A, X|*atomic*]
Action: 1) *Unfold* X

[You can't pull something out of an atomic; jitter.]
| **End Method** |

STEP 5.13: *Unfold* atomic ... end atomic

| **Method** UnfoldAtomic |

Goal: *Unfold* A|*atomic*
Action: 1) *Show* SEQUENTIAL-ORDERING(O|*ordering*, A)
2) *Show* SUPERFLUOUS_ATOMIC(A)
3) *Apply* UNFOLD-ATOMIC(A, O)

[You can unfold an atomic if you can show that there exists some valid sequential ordering of the statements and that no demonic or inferencing processes will be effected.]
| **End Method** |

We assume that the user verifies both conditions and the **atomic** is replaced with a **scoping_block**.

We must now find all places where the loop must be moved, i.e., all demons which trigger from the execution of **CREATE_PACKAGE**. The single location of interest is **RELEASE_PACKAGE_INTO_NETWORK**. After moving the maintenance code to that demon's response, we have the following:

```
demon RELEASE_PACKAGE_INTO_NETWORK(package.new)
  trigger package.new:LOCATED_AT = the source
  response
    begin
      loop (switch||MEMO_LOCATION_BIN(switch,package.new:DESTINATION))
        do update packages_due of PACKAGES_DUE_AT_SWITCH(switch,$)
           to PACKAGES_DUE_AT_SWITCH(switch,*) concat <package.new>;
      if LAST_PACKAGE_DESTINATION(*) ≠ package.new:DESTINATION
         then invoke WAIT[];
      update last_destination in LAST_PACKAGE_DESTINATION($)
           to package.new:DESTINATION
      update :LOCATED_AT of package.new
                           to (the source):SOURCE_OUTLET
    end;
```

We now have taken care of CREATE_PACKAGE, i.e., the initial increment of the sequences. We now must add code to decrement the sequences in appropriate cases.

The first step would be to maintain the sequence in RELEASE_PACKAGE_INTO_NETWORK: the update of the packages location to the source's outlet is a relevant change. However, since there is only one outlet pipe from the source, we can show that the maintenance code is unnecessary. The actual steps will be similar to the simplification of the maintenance code in CREATE_PACKAGE, and will be omitted here.

We will next look at the MOVE_PACKAGE demon since it updates the location of a package, and hence potentially can cause it to become misrouted or located at a switch.

```
demon MOVE_PACKAGE(package)
   trigger ∃ location.next || MOVEMENT_CONNECTION(package:LOCATED_AT,
                                                   location.next)
   response
     update :LOCATED_AT of package
        to MOVEMENT_CONNECTION(package:LOCATED_AT,*);
```

After inserting the necessary code ▸$_1$ to remove packages, we have:

```
demon MOVE_PACKAGE(package)
   trigger ∃ location.next || MOVEMENT_CONNECTION(package:LOCATED_AT,
                                                   location.next)
   response
     atomic
       update :LOCATED_AT of package
           to MOVEMENT_CONNECTION(package:LOCATED_AT,*);
▸1      loop switch ||
           ~(MEMO_LOCATION_BIN(switch package:DESTINATION)
                and
             ~(MOVEMENT_CONNECTION(package:LOCATED_AT,*) = switch)
                              asof everbefore)
                and
             (MEMO_LOCATION_BIN(MOVEMENT_CONNECTION(
                                        package:LOCATED_AT,*),
                                           package:DESTINATION)
                and
              ~∃ switch.2 ||
                   (MOVEMENT_CONNECTION(package:LOCATED_AT,*) =
                                                     switch.2
                      and
                    ~MEMO_LOCATION_BIN(switch.2:SWITCH_SETTING,
                                  package:DESTINATION)))))
         do update packages_due of PACKAGES_DUE_AT_SWITCH(switch,$)
          to PACKAGES_DUE_AT_SWITCH(switch,*) minus <package>
     end atomic;
```

Our only worry is if a package moves into a switch; if it moves to any other type of location, it cannot effect our sequence. When it moves into a switch, we must remove it from that switch sequence and possibly others if the switch is set wrong (because of bunching). Using a number of simplification steps (omitted here) we arrive at the following:

```
demon MOVE_PACKAGE(package)
   trigger ∃ location.next || MOVEMENT_CONNECTION(package:LOCATED_AT,
                                                        location.next)
   response
     atomic
       update :LOCATED_AT of package
             to MOVEMENT_CONNECTION(package:LOCATED_AT,*);
▸1      if
         ∃ switch.current ||
             (MOVEMENT_CONNECTION(package:LOCATED_AT,*) =
                                                        switch.current
                and
              MEMO_LOCATION_BIN(switch.current, package:DESTINATION))
       then
▸2       if MEMO_LOCATION_BIN(switch.current:SWITCH_SETTING,
                                          package:DESTINATION)
        then
▸3       update packages_due of PACKAGES_DUE_AT_SWITCH(switch.current,$)
            to PACKAGES_DUE_AT_SWITCH(switch.current,*) minus package
▸4       else
▸5         loop (switch||MEMO_LOCATION_BIN(switch,package:DESTINATION))
            do update packages_due of PACKAGES_DUE_AT_SWITCH(switch,$)
                 to PACKAGES_DUE_AT_SWITCH(switch,*) minus package;
       end atomic;
      end
```

To paraphrase, ▸$_1$ if a package is moved into a switch and that switch is on the route to the package's destination then: ▸$_2$ if the switch is set right then ▸$_3$ remove the package from the sequence due at the switch, else ▸$_4$ if the switch is set wrong then ▸$_5$ remove the package from all switches along the package's destination route, including the current one.

STEP 5.14: *Purify* if ... then ... in MOVE_PACKAGE

MOVE_PACKAGE is outside of our portion of the development, hence the introduced code must be moved in.

```
| Method  PurifyDemon                                                    |

        Goal: Purify A|action in D|demon
        Action: 1) Remove L from D

        [Remove unpure statement L from D.]
| End Method                                                             |
```

STEP 5.15: *Remove* $\blacktriangleright_1$ <u>if</u> ... <u>then</u> ... from MOVE_PACKAGE

```
| Method  RemoveFromDemon                                                |

        Goal: Remove A|action from D|demon
        Action: 1) Globalize A
                2) forall trigger-location[D2|demon, body-of[*, D], spec]
                      do Apply MOVE_STATEMENT_TO_DEMON(A, D2)

        [Find all demons that trigger from D and move the action A there.]
| End Method                                                             |
```

STEP 5.16: *Globalize* $\blacktriangleright_1$ <u>if</u> ... <u>then</u> ...

```
| Method  GlobalizeAction                                                |

        Goal: Globalize A|action
        Filter: a) component-of[A, X|atomic]
        Action: 1) Unfold X

        [You can't pull something out of an atomic; jitter.]
| End Method                                                             |
```

STEP 5.17: *Unfold* <u>atomic</u> ... <u>end</u> <u>atomic</u>

| **Method** UnfoldAtomic |

Goal: *Unfold* A|*atomic*

Action: 1) *Show* SEQUENTIAL-ORDERING(O|*ordering*, A)

2) *Show* SUPERFLUOUS_ATOMIC(A)

3) *Apply* UNFOLD-ATOMIC(A, O)

[You can unfold an atomic if you can show that there exists some valid sequential ordering of the statements and that no demonic or inferencing processes will be effected.]

| **End Method** |

We rely on the user to verify the two conditions. The actual unfolding uses the following transformation:

```
atomic
  update X:a to v;
  <expression using v>
end atomic
⇒
begin
  update X:a to v;
  <expression using X:a>
end
```

```
demon MOVE_PACKAGE(package)
   trigger ∃ location.next || MOVEMENT_CONNECTION(package:LOCATED_AT,
                                                          location.next)
   response
     begin
       update :LOCATED_AT of package
           to MOVEMENT_CONNECTION(package:LOCATED_AT,*);
       if
         ∃ switch.current | package:LOCATED_AT = switch.current
             and
         MEMO_LOCATION_BIN(switch.current, package:DESTINATION)
       then
         if MEMO_LOCATION_BIN(switch.current:SWITCH_SETTING,
                                                package:DESTINATION)
         then
         update packages_due of PACKAGES_DUE_AT_SWITCH(switch.current,$)
            to PACKAGES_DUE_AT_SWITCH(switch.current,*) minus package
         else
          loop (switch||MEMO_LOCATION_BIN(switch,package:DESTINATION))
          do update packages_due of PACKAGES_DUE_AT_SWITCH(switch,$)
                  to PACKAGES_DUE_AT_SWITCH(switch,*) minus package;
       end;
```

The maintenance code is now ready to be moved out of MOVE_PACKAGE. We must find all demons which trigger on the update of a package's location and move the unpure code to each. There are four demons to consider:

- MISROUTED_PACKAGE_REACHED_BIN
- SET_SWITCH
- PACKAGE_ENTERING_SENSOR
- PACKAGE_LEAVING_SENSOR

We will work on MISROUTED_PACKAGE_REACHED_BIN first.

```
demon MISROUTED_PACKAGE_REACHED_BIN(package, bin.reached, bin.intended)
  trigger package:LOCATED_AT = bin.reached
              and
          package:DESTINATION = bin.intended[62]
  response
     invoke MISROUTED_ARRIVAL(bin.reached, bin.intended)
```

After distributing the maintenance of PACKAGES_DUE_AT_SWITCH ▶$_1$ into the response of MISROUTED_PACKAGE_REACHED_BIN, we have the following:

```
demon MISROUTED_PACKAGE_REACHED_BIN(package, bin.reached, bin-intended)
  trigger package:LOCATED_AT = bin.reached
              and
          package:DESTINATION = bin.intended
  response
    begin
▶1     if
         ∃ switch.current | package:LOCATED_AT = switch.current
              and
         MEMO_LOCATION_BIN(switch.current, package:DESTINATION)
       then
         if MEMO_LOCATION_BIN(switch.current:SWITCH_SETTING,
                                                   package:DESTINATION)
         then
         update packages_due of PACKAGES_DUE_AT_SWITCH(switch.current,$)
            to PACKAGES_DUE_AT_SWITCH(switch.current,*) minus package
         else
          loop (switch||MEMO_LOCATION_BIN(switch,package:DESTINATION))
            do update packages_due of PACKAGES_DUE_AT_SWITCH(switch,$)
                  to PACKAGES_DUE_AT_SWITCH(switch,*) minus package;
     invoke MISROUTED_ARRIVAL(bin.reached, bin.intended)
   end
```

Since we know that *package* is located at a bin when this demon triggers, we can simplify away all of the newly added code since it relies on *package* being located at a switch.

Next, we will look at SET_SWITCH as we have developed it so far.

---

[62]Gist does not allow the same object to be bound to separate variables (see section 3).

```
demon SET_SWITCH(switch)
  trigger ∃ package ||
            package = first(PACKAGES_DUE_AT_SWITCH(* switch))
              and
            SWITCH_IS_EMPTY(switch)
  response
    begin
      update :SWITCH_SETTING of switch to
        (pipe || pipe = switch:SWITCH_OUTLET
                   and
               MEMO_LOCATION_BIN(pipe package:DESTINATION))
    end
```

Knowing that the package cannot be located at a switch when the maintenance code is executed allows us to employ a similar simplification process as on MISROUTED_PACKAGE_REACHED_BIN in getting rid of all of the introduced maintenance code (the actual steps are omitted here.).

The next location of interest is PACKAGE_LEAVING_SENSOR.

```
demon PACKAGE_LEAVING_SENSOR(package, sensor)
   trigger ~package:LOCATED_AT = sensor
   response null;
```

After unfolding the maintenance code, we have

```
demon PACKAGE_LEAVING_SENSOR(package, sensor)
   trigger ~package:LOCATED_AT = sensor
   response
▸1      if
         ∃ switch.current | package:LOCATED_AT = switch.current
             and
         MEMO_LOCATION_BIN(switch.current, package:DESTINATION)
       then
         if MEMO_LOCATION_BIN(switch.current:SWITCH_SETTING,
                                              package:DESTINATION)
         then
         update packages_due of PACKAGES_DUE_AT_SWITCH(switch.current,$)
            to PACKAGES_DUE_AT_SWITCH(switch.current,*) minus package
         else
          loop (switch||MEMO_LOCATION_BIN(switch,package:DESTINATION))
            do update packages_due of PACKAGES_DUE_AT_SWITCH(switch,$)
                  to PACKAGES_DUE_AT_SWITCH(switch,*) minus package;
```

We will return to simplify ▸$_1$ after a few more steps.

We have one location remaining to look at, PACKAGE_ENTERING_SENSOR.

```
demon PACKAGE_ENTERING_SENSOR(package, sensor)
   trigger package:LOCATED←AT = sensor
   response null;
```

After unfolding the maintenance code, we have

```
demon PACKAGE_ENTERING_SENSOR(package, sensor)
   trigger package:LOCATED←AT = sensor
   response
▸1     if
         ∃ switch.current | package:LOCATED_AT = switch.current
            and
         MEMO_LOCATION_BIN(switch.current, package:DESTINATION)
       then
         if MEMO_LOCATION_BIN(switch.current:SWITCH_SETTING,
                                                   package:DESTINATION)
         then
         update packages_due of PACKAGES_DUE_AT_SWITCH(switch.current,$)
            to PACKAGES_DUE_AT_SWITCH(switch.current,*) minus package
         else
          loop (switch||MEMO_LOCATION_BIN(switch,package:DESTINATION))
            do update packages_due of PACKAGES_DUE_AT_SWITCH(switch,$)
                 to PACKAGES_DUE_AT_SWITCH(switch,*) minus package;
```

We have now completed the distribution of maintenance code for PACKAGES←DUE←AT←SWITCH. However, there are several more optimizations we can perform. As a preliminary step, we will break out the supertype *sensor*. In the initial specification, the type *sensor* allowed several actions to be localized, and hence improved understanding. However, as a development progresses, abstractions such as *sensor* tend to get in the way and certain optimizations are made easier if they are removed. Such is the case here. The removal of *sensor* from several demons will allow us to further optimize the maintenance code introduced earlier. We will work on PACKAGE_LEAVING_SENSOR first.

STEP 5.18 *(user)*: *Casify* PACKAGE_LEAVING_SENSOR

```
| Method  CasifySuperTrigger                                    |

      Goal:  Casify D|demon
      Filter: a) trigger-of[T, D]
              b) component-of[S|supertype, T]
      Action: 1) Apply CASIFY_DEMON_SUPERTYPE(T, S)

      [Spawn a separate demon for every subtype X of S.]
| End Method                                                    |
```

We gain two new demons, only the first useful in the current environment[63].:

```
demon PACKAGE_LEAVING_SWITCH(package, switch)
   trigger ~package:LOCATED_AT = switch
   response
▸1      if
          ∃ switch.current | package:LOCATED_AT = switch.current
              and ...;

demon PACKAGE_LEAVING_BIN(package, bin)
   trigger ~package:LOCATED_AT = bin
   response
▸1      if
          ∃ switch.current | package:LOCATED_AT = switch.current
              and ...
```

Since the PACKAGE_LEAVING_SWITCH demon relies on a package not residing at a switch, the introduced code can be simplified away. Although the second demon, PACKAGE_LEAVING_BIN, is never triggered, we can expect that further elaboration of the spec will change this. In that case, we can simplify away the code by showing that the package's location after leaving a bin can never be a switch.

We next look at specializing *sensor* in PACKAGE_ENTERING_SENSOR.

STEP 5.19 *(user)*: *Casify* PACKAGE_ENTERING_SENSOR

```
| Method  CasifySuperTrigger                                        |

        Goal:  Casify D|demon
        Filter:  a) trigger-of[T, D]
                 b) component-of[S|supertype, T]
        Action:  1) Apply CASIFY_DEMON_SUPERTYPE(T, S)

        [Spawn a separate demon for every subtype X of S.]
| End Method                                                        |
```

[63] In the spec, a package currently never leaves a bin. Naturally, further elaboration of the spec will likely address issues of infinite capacity bins and what happens to packages after they reach a bin.

We gain two new demons.

```
demon PACKAGE_ENTERING_SWITCH(package, switch)
   trigger package:LOCATED_AT = switch
   response
▸1      if
          ∃ switch.current | package:LOCATED_AT = switch.current
              and
          MEMO_LOCATION_BIN(switch.current, package:DESTINATION)
        then
          if MEMO_LOCATION_BIN(switch.current:SWITCH_SETTING,
                                                      package:DESTINATION)
          then
          update packages_due of PACKAGES_DUE_AT_SWITCH(switch.current,$)
             to PACKAGES_DUE_AT_SWITCH(switch.current,*) minus package
          else
           loop (switch||MEMO_LOCATION_BIN(switch,package:DESTINATION))
             do update packages_due of PACKAGES_DUE_AT_SWITCH(switch,$)
                  to PACKAGES_DUE_AT_SWITCH(switch,*) minus package;

demon PACKAGE_ENTERING_BIN(package, bin)
   trigger package:LOCATED_AT = bin
   response
▸1      if
          ∃ switch.current | package:LOCATED_AT = switch.current
              and ...
```

We can get rid of the maintenance code from PACKAGE_ENTERING_BIN by showing that a package cannot be both at a bin and a switch.

Finally, we can do some minor simplification to PACKAGE_ENTERING_SWITCH.

```
demon PACKAGE_ENTERING_SWITCH(package, switch)
   trigger package:LOCATED_AT = switch
   response
       if
         MEMO_LOCATION_BIN(switch, package:DESTINATION)
       then
         if MEMO_LOCATION_BIN(switch:SWITCH_SETTING,
                                                   package:DESTINATION)
         then
          update packages_due of PACKAGES_DUE_AT_SWITCH(switch,$)
            to PACKAGES_DUE_AT_SWITCH(switch,*) minus package
         else
          loop (switch.1 || MEMO_LOCATION_BIN(switch.1,
                                                 package:DESTINATION))
          do update packages_due of PACKAGES_DUE_AT_SWITCH(switch.1,$)
               to PACKAGES_DUE_AT_SWITCH(switch.1,*) minus package;
```

This completes the maintenance of PACKAGES_DUE_AT_SWITCH. We have introduced code in RELEASE_PACKAGE_INTO_NETWORK to incrementally add packages to sequences and code in PACKAGE_ENTERING_SWITCH to do the corresponding removal.

C.6. Map Demons

At this point in the development, there are a number of demons defined in our portion of the specification:

1. RELEASE_PACKAGE_INTO_NETWORK
2. PACKAGE_ENTERING_SWITCH
3. PACKAGE_ENTERING_BIN
4. PACKAGE_LEAVING_SWITCH
5. PACKAGE_LEAVING_BIN
6. INIT_MEMO
7. SET_SWITCH
8. MISROUTED_PACKAGE_REACHED_BIN

There is nothing we can do with the first six since each triggers on an external event (e.g., packages entering the router, packages tripping sensors). However, the remaining two, SET_SWITCH and MISROUTED_PACKAGE_REACHED_BIN, need to be mapped. We will look first at SET_SWITCH.

STEP 6.1 *(user)*: *Map* SET_SWITCH

```
demon SET_SWITCH(switch)
  trigger ∃ package ||
▸1          package = first(PACKAGES_DUE_AT_SWITCH(* switch))
              and
▸2           SWITCH_IS_EMPTY(switch)
  response
    begin
      update :SWITCH_SETTING of switch to
          (pipe || pipe = switch:SWITCH_OUTLET and
                   MEMO_LOCATION_BIN(pipe package:DESTINATION))
    end
```

| Method CasifyDemon |

Goal: *Map* D|**demon**
Action: 1) *Casify* D
2) forall case-of[X, D] do *Map* X

[Try mapping by case analysis.]
| End Method |

STEP 6.2: *Casify* SET_SWITCH

SET_SWITCH may trigger on either of two events: ▸$_1$ a package becoming the first in some sequence due at a switch; ▸$_2$ a switch becoming empty. We will split the current SET_SWITCH demon into separate ones to trigger on each individually. Note that the selection of the trigger splitting method here requires a fair amount of insight. One has to notice that there are two components of the SET_SWITCH trigger, one that is under direct mechanical observation (a switch becoming empty) and one that is not (a package becoming the first of an internal sequence). The former may be handled by using existing sensing information while the latter will need to be maintained explicitly; two different development strategies will be required.

| Method CasifyConjunctiveTrigger |

Goal: *Casify* D|**demon**
Filter: a) gist-type-of[T|trigger-of[D], ***conjunction***]
Action: 1) *Show* INDIVIDUAL_START(D)
2) *Apply* SPLIT_CONJUNCTIVE_TRIGGER(D, T)

[It may be easier to break a demon up into special cases and then trying to map. Make sure that no new triggerings are created.]
| End Method |

Two new demons are spawned:

```
demon SET_SWITCH_WHEN_BUBBLE_PACKAGE(switch)
  trigger ∃ package ||
                package = first(PACKAGES_DUE_AT_SWITCH(* switch))
  response
    begin
      require  SWITCH_IS_EMPTY(switch) at ThisEvent)
      update :SWITCH_SETTING of switch to
         (pipe || pipe = switch:SWITCH_OUTLET and
                  MEMO_LOCATION_BIN(pipe package:DESTINATION))
    end

demon SET_SWITCH_ON_EXIT(switch)
  trigger SWITCH_IS_EMPTY(switch)
  response
    begin
      require (∃ package ||
             package = first(PACKAGES_DUE_AT_SWITCH(* switch))
                                  at ThisEvent)
      update :SWITCH_SETTING of switch to
         (pipe || pipe = switch:SWITCH_OUTLET and
                  MEMO_LOCATION_BIN(pipe package:DESTINATION))
    end
```

STEP 6.3: *Map* SET_SWITCH_WHEN_BUBBLE_PACKAGE

```
| Method  UnfoldDemon                                        |

      Goal: Map D|demon
      Action: 1) forall trigger-location[D, L, spec]
                          do Unfold D at L

      [To Map a demon, unfold it where appropriate.]
| End Method                                                 |
```

We must locate each place that the trigger may change, i.e., that PACKAGES_DUE_AT_SWITCH is changed. There are two such locations:

1. the sequence is incremented ▸$_1$ when a package enters the network (RELEASE_PACKAGE_INTO_NETWORK)

2. the sequence is decremented when a package enters a switch (PACKAGE_ENTERING_SWITCH).

We will look at the former first:

```
demon RELEASE_PACKAGE_INTO_NETWORK(package.new)
  trigger package.new:LOCATED_AT = the source
  response
    begin
      loop (switch||MEMO_LOCATION_BIN(switch,package.new:DESTINATION))
▶1       do update packages_due of PACKAGES_DUE_AT_SWITCH(switch,$)
            to PACKAGES_DUE_AT_SWITCH(switch,*) concat <package.new>;
      if LAST_PACKAGE_DESTINATION(*) ≠ package.new:DESTINATION
         then invoke WAIT[];
      update last_destination in LAST_PACKAGE_DESTINATION($)
           to package.new:DESTINATION;
      update :LOCATED_AT of package.new
           to (the source):SOURCE_OUTLET
    end;
```

STEP 6.4: *Unfold* **SET_SWITCH_WHEN_BUBBLE_PACKAGE at**

```
▶1     update packages_due of PACKAGES_DUE_AT_SWITCH(switch, $)
       to PACKAGES_DUE_AT_SWITCH(switch, *) concat <package.new>;
```

Method ScatterComputationOfDemon
Goal: *Unfold* D|**demon** at L
Filter: a) trigger-location[D, L, $]
Action: 1) *Apply* UNFOLD_DEMON_CODE(D L)
2) *Purify* L
[To unfold a demon D at a trigger point, stick in code to compute it and make sure L is within implementable portion of spec.]
End Method

After adding the maintenance code ▶$_2$, we have

```
demon RELEASE_PACKAGE_INTO_NETWORK(package.new)
  trigger package.new:LOCATED_AT = the source
  response
    begin
      loop (switch||MEMO_LOCATION_BIN(switch,package.new:DESTINATION))
        do
          begin
▸1          update packages_due of PACKAGES_DUE_AT_SWITCH(switch,$)
            to PACKAGES_DUE_AT_SWITCH(switch,*) concat <package.new>;
▸2          if ∃ package.1 ||
                ~((package.1 = first(PACKAGES_DUE_AT_SWITCH(switch,*))
                   asof last update of PACKAGES_DUE_AT_SWITCH(switch,$))
              and
                package.1 = first(PACKAGES_DUE_AT_SWITCH(switch,*))
            then
             begin
              require SWITCH_IS_EMPTY(switch)
              update :SWITCH_SETTING of switch to
                (pipe || pipe = switch:SWITCH_OUTLET and
                   MEMO_LOCATION_BIN(pipe package.1:DESTINATION))
             end
          end
      if LAST_PACKAGE_DESTINATION(*) ≠ package.new:DESTINATION
         then invoke WAIT[];
      update last_destination in LAST_PACKAGE_DESTINATION($)
           to package.new:DESTINATION
      update :LOCATED_AT of package.new
                        to (the source):SOURCE_OUTLET
    end;
```

In general, the unfolding of a demon with body B and trigger T at event E takes the following form:

```
<event E>     =>     <event E>
                     if ~T asof E and T (now) then B
```

In our case, E is the update of PACKAGES_DUE_AT_SWITCH and T is the trigger of SET_SWITCH_WHEN_BUBBLE_PACKAGE.

Some fairly sophisticated reasoning is needed to simplify further:

1. We know that this is the sole location where packages are added to sequences, and hence *package.new* was not part of the sequence in the previous state.

2. Given the semantics of sequence appending, we can reason that the only way that the first element of a sequence can change on an append is if the sequence was initially empty.

We require the user to supply much of the above reasoning; the system carries out the mundane portions (see example B, section E.14):

```
demon RELEASE_PACKAGE_INTO_NETWORK(package.new)
  trigger package.new:LOCATED_AT = the source
  response
    begin
      loop (switch||MEMO_LOCATION_BIN(switch,package.new:DESTINATION))
        do
         begin
          update packages_due of PACKAGES_DUE_AT_SWITCH(switch,$)
           to PACKAGES_DUE_AT_SWITCH(switch,*) concat <package.new>;
          if
             package.new = first(PACKAGES_DUE_AT_SWITCH(switch,*))
                 and
            SWITCH_IS_EMPTY(switch)
          then
             update :SWITCH_SETTING of switch to
               (pipe || pipe = switch:SWITCH_OUTLET and
                   MEMO_LOCATION_BIN(pipe package.new:DESTINATION))
         end
      if LAST_PACKAGE_DESTINATION(*) ≠ package.new:DESTINATION
         then invoke WAIT[];
      update last_destination in LAST_PACKAGE_DESTINATION($)
           to package.new:DESTINATION
      update :LOCATED_AT of package.new
                         to (the source):SOURCE_OUTLET
    end;
```

We will look next at PACKAGE_ENTERING_SWITCH.

```
demon PACKAGE_ENTERING_SWITCH(package, switch)
   trigger package:LOCATED_AT = switch
   response
       if
         MEMO_LOCATION_BIN(switch, package:DESTINATION)
       then
         if MEMO_LOCATION_BIN(switch:SWITCH_SETTING,
                                                     package:DESTINATION)
         then
▸1          update packages_due of PACKAGES_DUE_AT_SWITCH(switch,$)
             to PACKAGES_DUE_AT_SWITCH(switch,*) minus package
         else
          loop (switch.1||MEMO_LOCATION_BIN(switch.1,
                                                     package:DESTINATION))
▸2          do update packages_due of PACKAGES_DUE_AT_SWITCH(switch.1,$)
                   to PACKAGES_DUE_AT_SWITCH(switch.1,*) minus package;
```

Before preceding, we will factor the two updates of PACKAGES_DUE_AT_SWITCH ac▸$_1$,▸$_2$ into an **procedure** ▸$_3$ for the sake of conciseness.

STEP 6.5(user): *Factor*

```
      update packages_due of PACKAGES_DUE_AT_SWITCH(#switch[64], $)
          to PACKAGES_DUE_AT_SWITCH(#switch,*) minus #package
  in  PACKAGE_ENTERING_SWITCH
```

```
| Method  FactorDBMaintenanceIntoAction                          |

        Goal: Factor U|db-maintenance in L
        Action: 1) Apply CREATE_PROCEDURE_FROM_TEMPLATE(U A)
                2) forall pattern-match[U, W, L]
                     do Apply REPLACE_DBMAINTENACE_WITH_ACTION(W A)

        [Create a new procedure A and then find all matches W in L and replace each
        with a call to the new procedure A.]
| End Method                                                     |
```

[64] In a factor template, *#type.name* signifies a formal parameter. The # will be removed in the procedure definition.

```
demon PACKAGE_ENTERING_SWITCH(package, switch)
   trigger package:LOCATED_AT = switch
   response
       if
         MEMO_LOCATION_BIN(switch, package:DESTINATION)
       then
         if MEMO_LOCATION_BIN(switch:SWITCH_SETTING,
                                                    package:DESTINATION)
         then
           invoke TRIM_PACKAGES_DUE_AT_SWITCH(package, switch)
         else
           loop (switch.1||MEMO_LOCATION_BIN(switch.1,
                              package:DESTINATION))
             do invoke TRIM_PACKAGES_DUE_AT_SWITCH(package, switch.1)

▸3 procedure TRIM_PACKAGES_DUE_AT_SWITCH(package, switch)
           update packages_due of PACKAGES_DUE_AT_SWITCH(switch,$)
             to PACKAGES_DUE_AT_SWITCH(switch,*) minus package;
```

Now unfolding the maintenance code for SET_SWITCH_WHEN_BUBBLE_PACKAGE ▸4 into the newly created procedure, we have

```
demon PACKAGE_ENTERING_SWITCH(package, switch)
    trigger package:LOCATED_AT = switch
    response
        if
           MEMO_LOCATION_BIN(switch, package:DESTINATION)
        then
           if MEMO_LOCATION_BIN(switch:SWITCH_SETTING,
                                                    package:DESTINATION)
           then invoke TRIM_PACKAGES_DUE_AT_SWITCH(package,
                                                      switch.current)
           else
             loop (switch||MEMO_LOCATION_BIN(switch,package:DESTINATION))
               do invoke TRIM_PACKAGES_DUE_AT_SWITCH(package, switch);

procedure TRIM_PACKAGES_DUE_AT_SWITCH(package, switch)
     begin
          update packages_due of PACKAGES_DUE_AT_SWITCH(switch,$)
              to PACKAGES_DUE_AT_SWITCH(switch,*) minus package;
▸4         if
            ∃ package.1 ||
            ~((package.1 = first(PACKAGES_DUE_AT_SWITCH(switch,*))
               asof last update of PACKAGES_DUE_AT_SWITCH(switch, $))
             and
            package.1 = first(PACKAGES_DUE_AT_SWITCH(switch,*))
          then
              begin
                require  SWITCH_IS_EMPTY(switch)
                update :SWITCH_SETTING of switch to
                  (pipe || pipe = switch:SWITCH_OUTLET and
                      MEMO_LOCATION_BIN(pipe, package.1:DESTINATION))
               end
     end
```

Note that the factoring was a mixed blessing. While it did allow us to unfold in a single place, it prevents us from carrying out some further optimization: if the procedure is being called when the switch is set right, we can safely ignore the switch setting code (we can show that the switch is non-empty). To actually get rid of this unneeded case, we will eventually have to unfold the procedure back into the demon and simplify.

We can simplify the procedure further if we rely on the user to supply the following necessary reasoning step: the only way for a new package to become the first of the sequence is by the removal of the head of the sequence.

```
procedure TRIM_PACKAGES_DUE_AT_SWITCH(package, switch)
    begin
        if first(PACKAGES_DUE_AT_SWITCH(switch, *) = package
        then
         begin
            update packages_due of PACKAGES_DUE_AT_SWITCH(switch,$)
              to PACKAGES_DUE_AT_SWITCH(switch,*) minus package;
            begin
              require SWITCH_IS_EMPTY(switch)
              update :SWITCH_SETTING of switch to
                (pipe || pipe = switch:SWITCH_OUTLET and
                   MEMO_LOCATION_BIN(pipe,
                         first(PACKAGES_DUE_AT_SWITCH(switch, *)
                                   ):DESTINATION))
             end
         end
        else
            update packages_due of PACKAGES_DUE_AT_SWITCH(switch,$)
              to PACKAGES_DUE_AT_SWITCH(switch,*) minus package;
    end
```

This takes care of the SET_SWITCH_WHEN_BUBBLE_PACKAGE demon which deals with the package sequence changing. We now must take care of setting a switch when it becomes empty, an event captured by the SET_SWITCH_ON_EXIT demon.

```
demon SET_SWITCH_ON_EXIT(switch)
  trigger SWITCH_IS_EMPTY(switch)
  response
    begin
       require (∃ package ||
            package = first(PACKAGES_DUE_AT_SWITCH(* switch))
                           at ThisEvent)
       update :SWITCH_SETTING of switch to
          (pipe || pipe = switch:SWITCH_OUTLET and
                MEMO_LOCATION_BIN(pipe package:DESTINATION))
    end
```

STEP 6.6: *Map* SET_SWITCH_ON_EXIT

Instead of unfolding this demon as we did with SET_SWITCH_WHEN_BUBBLE_PACKAGE,

we will attempt to consolidate it with an already existing demon, PACKAGE_LEAVING_SWITCH.

```
demon PACKAGE_LEAVING_SWITCH(package, switch)
▶1  trigger ~package:LOCATED_AT = switch
    response null;

demon SET_SWITCH_ON_EXIT(switch)
▶2 trigger SWITCH_IS_EMPTY(switch)
   response
     begin
        require (∃ package ||
               package = first(PACKAGES_DUE_AT_SWITCH(* switch))
                              at ThisEvent)
        update :SWITCH_SETTING of switch to
           (pipe || pipe = switch:SWITCH_OUTLET and
                    MEMO_LOCATION_BIN(pipe package:DESTINATION))
     end

▶3 relation SWITCH_IS_EMPTY(SWITCH)
   definition  not exists package || package:located_at = switch;
```

| Method MapByConsolidation |

Goal: Map D|demon

Filter: a) pattern-match[demon, D2, spec]

b) D ≠ D2

Action: 1) *Consolidate* D and D2

[To map D, find some other demon D2 and consolidate.]

| End Method |

Naturally, the selection of the right demon to consolidate with is crucial.

STEP 6.7: *Consolidate* SET_SWITCH_ON_EXIT and PACKAGE_LEAVING_SWITCH

| Method MergeDemons |

Goal: *Consolidate* D1|**demon** and D2|**demon**

Action: 1) *Equivalence* trigger-of[D1] and trigger-of[D2]

2) *Equivalence* var-declaration-of[D1] and var-declaration-of[D2]

3) *Show* MERGEABLE_DEMONS(D1, D2, I|**ordering**)

4) *Apply* DEMON_MERGE(D1, D2, I)

[You can consolidate two demons if you can show that they have the same local variables, the same triggering pattern and that they meet certain merging conditions.]

| End Method |

STEP 6.8: *Equivalence*

▸1 trigger *~package*:LOCATED_AT = *switch*
▸2 trigger SWITCH_IS_EMPTY(*switch*)

As in step 2.3, we will anchor the first trigger and try to reformulate the second.

| Method Anchor1 |

Goal: *Equivalence* X *and* Y

Action: 1) *Reformulate* Y *as* X

[Try changing the second construct into something that matches the first.]

| End Method |

STEP 6.9: *Reformulate* SWITCH_IS_EMPTY(*switch*) as

~package:LOCATED_AT = *switch*

| Method ReformulateDerivedRelation |

Goal: *Reformulate* RR|***relation-reference*** as X
Filter: a) gist-type-of[name-of[R, RR],
derived-relation]
Action: 1) *Unfold* R at RR

[Try reformulating the body as X.]

| End Method |

STEP 6.10: *Unfold* ▸$_3$ SWITCH_IS_EMPTY at reference ▸$_2$

| Method ScatterComputationOfDerivedRelation |

Goal: *Unfold* DR|***derived-relation*** at L
Filter: a) reference-location[DR, L, S]
Action: 1) *Apply* UNFOLD_COMPUTATION_CODE(DR L)
2) *Purify* L

[To unfold a derived relation DR at a reference point, stick in code to compute it and make sure L is within implementable portion of spec.]

| End Method |

The unfolding of SWITCH_IS_EMPTY still does not achieve the reformulation goal in step 6.9, hence it is reposted:

STEP 6.11 ***(reposted)*:** *Reformulate*

trigger ~∃ *package.0* || *package.0*:LOCATED_AT = *switch*
as trigger ~*package*:LOCATED_AT = *switch*

Our goal here is to produce a more general trigger for SWITCH←IS←EMPTY than its current one. That is, we want to trigger whenever a package is no longer located at a switch no matter if a new package has moved into the switch or not. The current trigger requires that a package leave a switch and that no other switch moves in immediately behind it.

```
| Method  ReformulateExistentialTrigger                              |

      Goal: Reformulate T|trigger ~∃ o||R(o)  as R(o')
      Action: 1) Show TRIGGER_GENERALIZABLE(T)
              2) Apply GENERALIZE_TRIGGER(T)

      [You can reformulate an existential trigger into a universally quantified one
      under certain conditions.]
| End Method                                                         |
```

We assume the user verifies that the trigger is generalizable. After application of GENERALIZE_TRIGGER, we have

```
demon PACKAGE_LEAVING_SWITCH(package, switch)
▸1  trigger ~package:LOCATED_AT = switch
   response null;

demon SET_SWITCH_ON_EXIT(package.gen, switch)
▸2 trigger ~package.gen:LOCATED_AT = switch
  response
   if ~∃ package||package:LOCATED_AT = switch
    then begin
       require (∃ package ||
           package = first(PACKAGES_DUE_AT_SWITCH(* switch))
                          at ThisEvent)
       update :SWITCH_SETTING of switch to
          (pipe || pipe = switch:SWITCH_OUTLET and
                  MEMO_LOCATION_BIN(pipe package:DESTINATION))
    end
```

STEP 6.12: *Equivalence (package, switch) and (package.gen, switch)*

The same renaming strategy (with the exception of using Anchor2 in place of Anchor1) used in step 2.10 will be used; we omit the steps here.

After consolidation, we have

```
demon PACKAGE_LEAVING_SWITCH(package.gen, switch)
   trigger ~package.gen:LOCATED_AT = switch
   response
    if ~∃ package||package:LOCATED_AT = switch
     then begin
       require (∃ package ||
            package = first(PACKAGES_DUE_AT_SWITCH(* switch))
                                          at ThisEvent)
       update :SWITCH_SETTING of switch to
          (pipe || pipe = switch:SWITCH_OUTLET and
                   MEMO_LOCATION_BIN(pipe package:DESTINATION))
     end
```

This finishes our task of mapping away SET_SWITCH.

STEP 6.13*(user)*: *Map* MISROUTED_PACKAGE_REACHED_BIN

```
demon MISROUTED_PACKAGE_REACHED_BIN(package, bin.reached, bin.intended)
  trigger package:LOCATED_AT = bin.reached
                  and
          package:DESTINATION = bin.intended
  response invoke MISROUTED_ARRIVAL(bin.reached, bin.intended)
```

```
| Method  CasifyDemon                                   |

        Goal: Map D|demon
        Action: 1) Casify D
                2) forall case-of[X, D] do Map X

        [Try mapping by case analysis.]
| End Method                                            |
```

STEP 6.14: *Casify* MISROUTED_PACKAGE_REACHED_BIN

We will use the same trigger splitting strategy as used on SET_SWITCH in the previous

section. **MISROUTED_PACKAGE_REACHED_BIN** may trigger on either of two events: a package becoming located at a bin; a package's destination being set. The selection of the trigger splitting method here requires the same insight as in the **SET_SWITCH** case: one has to notice that one of the two components of the trigger is under direct mechanical observation (a switch entering a bin) and one is not (a package's destination changing).

```
| Method  CasifyConjunctiveTrigger                                  |

        Goal: Casify D|demon
        Filter: a) gist-type-of[T|trigger-of[D],
                                        conjunction]
        Action: 1) Show INDIVIDUAL_START(D)
                2) Apply SPLIT_CONJUNCTIVE_TRIGGER(D, T)

        [It may be easier to break a demon up into special cases and then trying to
        map. Make sure that no new triggerings are created.]
| End Method                                                        |
```

Two new demons are spawned:

```
demon MISROUTED_PACKAGE_LOCATED_AT_BIN(package,bin.reached,bin-intended)
  trigger package:LOCATED_AT = bin.reached
  response
     begin
          require (package:DESTINATION = bin.intended
                      at ThisEvent);
          invoke MISROUTED_ARRIVAL(bin.reached, bin.intended)
     end;

demon MISROUTED_PACKAGE_DESTINATION_SET(package,bin.reached,bin-intended)
  trigger package:DESTINATION = bin.intended
  response
     begin
       require (package:LOCATED_AT = bin.reached
                      at ThisEvent);
       invoke MISROUTED_ARRIVAL(bin.reached, bin.intended)
     end;
```

STEP 6.15: *Map* **MISROUTED_PACKAGE_LOCATED_AT_BIN**

```
| Method  MapByConsolidation                                    |

      Goal: Map D|demon
      Filter: a) pattern-match[demon, D2, spec]
              b) D ≠ D2
      Action: 1) Consolidate D and D2

      [To map D, find some other demon D2 and consolidate.]
| End Method                                                    |
```

STEP 6.16: *Consolidate* MISROUTED_PACKAGE_LOCATED_AT_BIN and PACKAGE_ENTERING_BIN

```
demon PACKAGE_ENTERING_BIN(package, bin)
   trigger package:LOCATED_AT = bin
   response null;
```

```
| Method  MergeDemons                                           |

      Goal: Consolidate D1|demon and D2|demon
      Action: 1) Equivalence trigger-of[D1] and
                                 trigger-of[D2]
              2) Equivalence var-declaration-of[D1] and
                                 var-declaration-of[D2]
              3) Show MERGEABLE_DEMONS(D1, D2, I|ordering)
              4) Apply DEMON_MERGE(D1, D2, I)

      [You can consolidate two demons if you can show that they have the same
      local variables, the same triggering pattern and that they meet certain
      merging conditions.]
| End Method                                                    |
```

STEP 6.17: *Equivalence* (*package, bin.reached, bin.intended*) and (*package, bin*)

```
| Method  EquivalenceCompoundStructures2                                   |

        Goal: Equivalence  S1|compound-structure and
                           S2|compound-structure
        Filter: a) gist-type-of[*, S1] = gist-type-of[*, S2]
                b) ~fixed-structure[S1]
                c) component-correspondence[S1, S2, C|correspondence]
        Action: 1) forall correspondence-pairs[C, C1, C2]
                     do Equivalence C1 and C2

        {Divide-and-conquer: make the components of two non-fixed structures
        equivalent.}
| End Method                                                               |
```

Choosing the correct correspondence here is a little tricky. Being of the same type, the two *package* variables are paired-off. However, *bin* can be paired with either *bin.reached* or *bin.intended*. We note that both *bin* and *bin.reached* occur in their respective triggers and use this clue to make the right choice.

STEP 6.18: *Equivalence bin.reached* and *bin*

As in step 2.10, we will eventually anchor the first and then rename.

Our equivalence goal from step 6.17 is still not achieved and hence is reposted.

STEP 6.19*(reposted)*: *Equivalence* (*package, bin.reached, bin.intended*) and (*package, bin.reached*)

Reapplying EquivalenceCompoundStructures2 now will gain us nothing. We try a new method.

```
| Method  AddNewVar                                                    |

        Goal: Equivalence L1|variable-list and L2|variable-list
        Filter: a) length[L1] > length[L2]
                b) member[V|variable-declaration, L1]
                c) ~member[V, L2]
        Action: 1) Show INTRODUCEABLE-VAR-NAME(V, L2)
                2) Apply INTRODUCE-NEW-VAR(V, L2)

        [Try adding a new var to make the two lists equivalent.]
| End Method                                                           |
```

After consolidation, we have

```
demon PACKAGE_ENTERING_BIN(package, bin.reached, bin.intended)
  trigger package:LOCATED_AT = bin.reached;
  response
     begin
       require (package:DESTINATION = bin.intended
                     at ThisEvent);
       invoke MISROUTED_ARRIVAL(bin.reached, bin.intended)
     end;
```

We next must take care of MISROUTED_PACKAGE_DESTINATION_SET.

STEP 6.20: *Map* MISROUTED_PACKAGE_DESTINATION_SET

```
| Method  UnfoldDemon                                                  |

        Goal: Map D|demon
        Action: 1) forall trigger-location[D, L, spec]
                          do Unfold D at L

        [To Map a demon, unfold it where appropriate.]
| End Method                                                           |
```

We must locate each place that a package's destination is changed. The single such location is at CREATE_PACKAGE.

```
demon CREATE_PACKAGE()
    trigger RANDOM()
    response
       atomic
         create package.new ||
             package.new:DESTINATION = a bin and
             package.new:LOCATED_AT = the source;
```

STEP 6.21: *Unfold* MISROUTED_PACKAGE_DESTINATION_SET at

```
create package.new ||
 package.new:DESTINATION = a bin and
 package.new:LOCATED_AT = the source;
```

```
| Method  ScatterComputationOfDemon                              |

      Goal: Unfold D|demon at L
      Filter: a) trigger-location[D, L, $]
      Action: 1) Apply UNFOLD_DEMON_CODE(D L)
              2) Purify L

      [To unfold a demon D at a trigger point, stick in code to compute it and make
      sure L is within implementable portion of spec.]
| End Method                                                     |
```

After adding the maintenance code, we have

```
demon CREATE_PACKAGE()
    trigger RANDOM()
    response
     begin
       atomic
         create package.new ||
             package.new:DESTINATION = a bin and
             package.new:LOCATED_AT = the source;
       end atomic
       if ∃ bin.intended, bin.reached ||
              ~((package.new:DESTINATION = bin.intended)
                     asof last update of package.new:DESTINATION)
                  and
              package.new:DESTINATION = bin.intended
       then
         begin
           require package.new:LOCATED_AT = bin.reached;
           invoke MISROUTED_ARRIVAL(bin.reached, bin.intended)
         end
     end
```

By showing that the require statement is always false, we can remove the MISROUTED_ARRIVAL procedure and finally the entire newly introduced conditional, leaving CREATE_PACKAGE in its original state.

C.7. Termination State

This ends our development of the package router. The state of the router at this point is given below. The Gist/TI group is currently working on an intermediate-level language called WILL which is able to implement directly this form of program.

Portions which have not changed from the initial spec given in Appendix A are:

- type hierarchy, including attributes (*sensor* could be removed since it is no longer referenced)
- constraints
  - MORE_THAN_ONE_SOURCE
  - PIPE_EMERGES_FROM_UNIQUE_SWITCH_OR_BIN
  - UNIQUE_PIPE_LEADS_TO_SWITCH_OR_BIN
  - SOURCE_ON_ROUTE_TO_ALL_BINS
- relations
  - MISROUTED
  - SWITCH_IS_EMPTY
- demons
  - CREATE_PACKAGE
  - MOVE_PACKAGE
- procedure
  - MISROUTED_ARRIVAL

Portions of the specification which are new or have changed are given below.

```
demon RELEASE_PACKAGE_INTO_NETWORK(package.new)
  trigger package.new:LOCATED_AT = the source
  response
    begin
      loop (switch||MEMO_LOCATION_BIN(switch,package.new:DESTINATION))
        do
         begin
          update packages_due of PACKAGES_DUE_AT_SWITCH(switch,$)
           to PACKAGES_DUE_AT_SWITCH(switch,*) concat <package.new>;
          if
             package.new = first(PACKAGES_DUE_AT_SWITCH(switch,*))
                 and
           SWITCH_IS_EMPTY(switch)
          then
             update :SWITCH_SETTING of switch to
               (pipe || pipe = switch:SWITCH_OUTLET and
                  MEMO_LOCATION_BIN(pipe package.new:DESTINATION))
         end
      if LAST_PACKAGE_DESTINATION(*) ≠ package.new:DESTINATION
         then invoke WAIT[];
      update last_destination in LAST_PACKAGE_DESTINATION($)
           to package.new:DESTINATION
      update :LOCATED_AT of package.new
                        to (the source):SOURCE_OUTLET
    end;
```

```
demon PACKAGE_ENTERING_SWITCH(package, switch)
   trigger package:LOCATED_AT = switch
   response
      if
        MEMO_LOCATION_BIN(switch, package:DESTINATION)
      then
        if MEMO_LOCATION_BIN(switch:SWITCH_SETTING,
                                              package:DESTINATION)
        then invoke TRIM_PACKAGES_DUE_AT_SWITCH(package,
                                                   switch.current)
        else
         loop (switch||MEMO_LOCATION_BIN(switch,package:DESTINATION))
           do invoke TRIM_PACKAGES_DUE_AT_SWITCH(package, switch);
```

```
procedure TRIM_PACKAGES_DUE_AT_SWITCH(package, switch)
    begin
        if first(PACKAGES_DUE_AT_SWITCH(switch, *)) = package
        then
         begin
            update packages_due of PACKAGES_DUE_AT_SWITCH(switch,$)
              to PACKAGES_DUE_AT_SWITCH(switch,*) minus package;
            begin
              require SWITCH_IS_EMPTY(switch)
              update :SWITCH_SETTING of switch to
                (pipe || pipe = switch:SWITCH_OUTLET and
                   MEMO_LOCATION_BIN(pipe,
                        first(PACKAGES_DUE_AT_SWITCH(switch, *)
                                  ):DESTINATION))
             end
         end
        else
            update packages_due of PACKAGES_DUE_AT_SWITCH(switch,$)
              to PACKAGES_DUE_AT_SWITCH(switch,*) minus package;
    end
```

```
demon PACKAGE_LEAVING_SWITCH(package.gen, switch)
   trigger ~package.gen:LOCATED_AT = switch
   response
    if ~∃ package||package:LOCATED_AT = switch
     then begin
       require (∃ package ||
            package = first(PACKAGES_DUE_AT_SWITCH(* switch))
                            at ThisEvent)
       update :SWITCH_SETTING of switch to
          (pipe || pipe = switch:SWITCH_OUTLET and
                  MEMO_LOCATION_BIN(pipe package:DESTINATION))
     end
```

```
demon PACKAGE_ENTERING_BIN(package, bin.reached, bin.intended)
  trigger package:LOCATED_AT = bin.reached;
  response
    begin
      require (package:DESTINATION = bin.intended
                    at ThisEvent);
      invoke MISROUTED_ARRIVAL(bin.reached, bin.intended)
    end;
```

```
demon PACKAGE_LEAVING_BIN(package, bin)
   trigger ~package:LOCATED_AT = bin
   response null;
```

```
relation LAST_PACKAGE_DESTINATION(last_destination| bin);

relation PACKAGES_DUE_AT_SWITCH(packages_due|sequence of package,
                                            switch);

relation MEMO_LOCATION_BIN(location, bin);
```

```
relation MEMO_LOCATION_BIN(location, bin);

demon INITIALIZE_MEMO_LOCATION_BIN()
   trigger: (start initialization_state)
   response
    begin
      loop B | BIN do insert MEMO_LOCATION_BIN(B, B);
      loop L | LOCATION ||
                      MEMO_LOCATION_BIN(L, B) and
                      L = L2:CONNECTION_TO_SWITCH_OR_BIN
             do insert MEMO_LOCATION_BIN(L2, B);
    end
```

Appendix D
Method Selection Overlay

This appendix presents the selection information used to produce the router development in appendix C. When overlayed with the development, the complete problem solving trace is explicated. The sectioning follows that of C. Each step here has the following form:

Step i.j: *abbreviated development goal*

<u>**Candidate Set**</u>
[<augmented method>]^0

➤ *General Rules*: [<general selection rule>]^0

➤ *Method Specific Rules*: [<method specific rule>]^0

➤ *Resource Rules*: [<resource rule>]^0

➤ *Ordering Rules*: [<ordering rule>]^0

<u>**Method Ordering**</u>: [<ordered method list>]^0

➤ *Action Ordering Rules*: [<action ordering rule>]^0

Comment: *Optional comments on interesting problem solving features of the step.*

An <augmented method> under the Candidate Set has the following form:

[Abrev:] MethodName [(<opinion> SelectionRule)]^0

An <opinion> is either a signed weight in the case where SelectionRule is a non-ordering rule or an ordering operator (i.e. >,<) for ordering rules. In the latter case, (< Foo) says that the current method has been ordered after some other method or set of methods by selection rule Foo. To find the method or meohds which are ordered before this method, look for the corresponding (> Foo).

If a candidate method contains unbound free varaibles, then a breakout of all instantiated bindings is given under the MethodName (see for example, step 1.2). Each instantiation has the following form:

[Abrev:] Binding [(<Opinion> SelectionRule)]^0

Note that opinions expressed about the general MethodName are inherited by any of its particular bound instantiaions.

A list of the selection rules augmenting the candidate set is brokenout by type below the Candidate Set. This is redudant information provided for convenience.

Finally, <ordered method list> is a partial ordering of the Candidate Set with the following form:

$\text{MethodSet}_1\text{(Sum),..MethodSet}_n\text{(Sum)}$

A MethodSet is either a 1) single method or 2) a group of MethodSets from the Candidate Set. In the second case, the set is marked off by set brackets ({ }). After each single method is the sum of all weights provided by the selection rules. If no weight-giving rules fired then a dash appears in place of the sum. If MethodSet_i occurs before MethodSet_j in the list then all methods in MethodSet_i are rated more highly than all methods of MethodSet_j. Methods within a MethodSet have the same rating.

Not all methods of the Candidate Set may appear in the ordering list. If a method's weighted sum is below a certain threshold, 1 currently, it will not appear. Also, if method M1 is ordered by a selection rule after method M2 whose sum is below the theshold, M1 will not appear, no matter what its sum is. Currently, methods which have no ordering information associated with them are included last in the list.

Bold facing is used in the <method order list> to mark the method actually chosen in the router development. Bold faced methods which do not appear first in the list represent locations where one or more alternative methods were rated more highly thatn the method finally chosen.

The details of the Glitter selection engine are discussed more fully in chapter 7.

D.1. Remove PACKAGES_EVER_AT_SOURCE

Step 1.1:*(user)* Remove peas (packages_ever_at_source) from spec

Candidate Set

☐ RR: RemoveRelation (+2 BurnedOutHulk) (+2 *RemoveRelation1)

➤ *General Rules*: BurnedOutHulk

➤ *Method Specific Rules*: *RemoveRelation1

Method Ordering: RR(+4)

Step 1.2: *Remove* reference to peas from spec

Candidate Set

☐ BabyWithBathWater

- BWBW1: Y bound to ***relative-retrieval*** (-2 *BabyWithBathWater3)
- BWBW2: Y bound to ***derived-object*** (-2 *BabyWithBathWater3)
- BWBW3: Y bound to ***conditional*** (0 *BabyWithBathWater1)
- BWBW4: Y bound to ***demon*** (-1 *BabyWithBathWater2)

☐ MegaMove (+1 FillIn) (> RemoveRef1)

- MM1: Y bound to ***relative-retrieval*** (+2 *MegaMove1) (< RemoveRef2)
- MM2: Y bound to ***derived-object*** (+2 *MegaMove1) (> RemoveRef2)

☐ PositionalMegaMove (+1 FillIn) (< RemoveRef1)

- PMM1: Y bound to ***relative-retrieval*** (+1 *PositionalMegaMove) (< RemoveRef3)
- PMM2: Y bound to ***derived-object*** (+1 *PositionalMegaMove) (> RemoveRef3)

☐ RemoveByObjectizingContext

- RBOC1: Y bound to ***relative-retrieval***
- RBOC2: Y bound to ***derived-object***

➤ *General Rules*: FillIn

➤ *Method Specific Rules*: *BabyWithBathWater, *MegaMove1, *PositionalMegaMove

➤ *Ordering Rules*: RemoveRef1, RemoveRef2, RemoveRef3

Method Ordering: MM2(+3), MM1(+3), PMM2(+2), PMM1(+2), {RBOC1(-), RBOC2(-)}

Step 1.3: *Isolate* derived object

<u>Candidate Set</u>

☐ FGIR: FoldGenericIntoRelation (+ 2 *FoldGenericIntoRelation)

➤ *Method Specific Rules*: *FoldGenericIntoRelation

<u>Method Ordering</u>: FGIR(+ 2)

Step 1.4: *Globalize* derived object

<u>Candidate Set</u>

☐ GDO: GlobalizeDerivedObject (+ 2 *GlobalizeDerivedObject)

➤ *Method Specific Rules*: *GlobalizeDerivedObject

<u>Method Ordering</u>: GDO

Step 1.5: (try) *Reformulate* p.new as global

<u>Candidate Set</u>:

☐ ReformulateLocalAsFirst (+ 2 ReformulateLocalAsSequenceExpression) (< ReformLoc2)

* RLAF: R bound to packages_ever_at_source

☐ ReformulateLocalAsLast (+ 2 ReformulateLocalAsSequenceExpression) (> ReformLoc2)

* RLAL: R bound to packages_ever_at_source

➤ *General Rules*: ReformulateLocalAsSequenceExpression

➤ *Ordering Rules*: ReformLoc2

<u>Method Ordering</u>: RLAF(+ 2), RLAL(+ 2)

Step 1.6: *Reformulate* p.new as <u>last</u>(peas(*))

<u>Candidate Set</u>

☐ ∅

no rules fired

Step 1.7:(*user*) <u>*Manual*</u> manual-replace(p.new last(peas))

<u>Candidate Set</u>

☐ manual step

no rules fired

Step 1.8: *MaintainIncrementally* previous_package

<u>Candidate Set</u>

- ☐ SMFDR: ScatterMaintenanceForDerivedRelation (+2 *ScatterMaintenanceForDerivedRelation)

➤ *Method Specific Rules*: *ScatterMaintenanceForDerivedRelation

<u>Method Ordering</u>: SMFDR(+2)

Step 1.9: *Flatten* previous_package

<u>Candidate Set</u>

- ☐ Flatten (+2 *Flatten)

➤ *Method Specific Rules*: *Flatten

<u>Method Ordering</u>: Flatten(+2)

Step 1.10: *Map* peas

<u>Candidate Set</u>

- ☐ MDR: MaintainDerivedRelation (+2 *MDR)
- ☐ UDR: UnfoldDerivedRelation (+2 *UnfoldDerivedRelation1) (-2 MapSubOfRemove2)

➤ *General Rules*: MapSubOfRemove2

➤ *Method Specific Rules*: *MaintainDerivedRelation, *UnfoldDerivedRelation1

<u>Method Ordering</u>: MDR(+2)

Comment: *Normally,the methods for maintaining and unfolding a derived relation compete equally. However, the general rule MapSubOfRemove recognizies certain contexts in which scattering what is currently a global definition may lead to difficulties further along in the development, i.e. if we are trying to remove a relation then scattering references to it througout the program is a non-cooperating strategy.*

Step 1.11: *MaintainIncrementally* peas

<u>Candidate Set</u>

- ☐ ISMD: IntroduceSeqMaintenanceDemon (+1 DemonsAreGood) (+1 MapSubOfRemove1) (+1 ReadyToGo) (+1 ReformUnnecessary)
- ☐ SMFDR: ScatterMaintenanceForDerivedRelation (-2 MapSubOfRemove2) (+2 *SMFDR)

➤ *General Rules*: DemonsAreGood, MapSubOfRemove1, MapSubOfRemove2

➤ *Method Specific Rules*: *ScatterMaintenanceForDerivedRelation

➤ *Resource Rules*: ReformUnnecessary, ReadyToGo

<u>Method Ordering</u>: ISMD(+4)

Step 1.12: *Remove* reference peas from spec

Candidate Set

☐ BabyWithBathWater

- BWBW1: Y bound to ***relative-retrieval*** (-2 •BabyWithBathWater3)
- BWBW2: Y bound to ***derived-object*** (-2 •BabyWithBathWater3)
- BWBW3: Y bound to ***update*** (-2 •BabyWithBathWater3)
- BWBW4: Y bound to ***atomic*** (-2 •BabyWithBathWater3)
- BWBW5: Y bound to ***demon*** (-1 •BabyWithBathWater2)

☐ MegaMove (+1 FillIn)

- MM1: Y bound to ***relative-retrieval*** (+2 •MegaMove1) (< RemoveRef2)
- MM2: Y bound to ***derived-object*** (-2 •MegaMove2) (> RemoveRef2)

☐ PositionalMegaMove (+1 FillIn)

- PMM1: Y bound to ***relative-retrieval*** (+1 •PositionalMegaMove) (< RemoveRef3)
- PMM2: Y bound to ***derived-object*** (+1 •PositionalMegaMove) (> RemoveRef3)

☐ RemoveByObjectizingContext

- RBOC1: Y bound to ***relative-retrieval***
- RBOC2: Y bound to ***derived-object***

☐ ReplaceRefWithValue (+1 FillIn) (-2 •ReplaceRefWithValue2)

➤ *General Rules*: FillIn

➤ *Method Specific Rules*: •MegaMove1, •MegaMove2, •BabyWithBathWater, •PositionalMegaMove, •ReplaceRefWithValue2

➤ *Ordering Rules*: RemoveRef2, RemoveRef3

Method Ordering: PMM2(+2), PMM1(+2), {RBOC1(-), RBOC2(-)}

Step 1.13: *Reformulate* derived-object as ***positional-retrieval***

Candidate Set

☐ RDO: ReformulateDerivedObject (+2 •ReformulateDerivedObject)

➤ *Method Specific Rules*: •ReformulateDerivedObject

Method Ordering: RDO(+2)

Comment: *Note that it's up to the user to determine "close to" here, i.e. he must determine if the body of the derived object, a relatinal retrieval, can be changed into a positional one.*

Step 1.14: *Reformulate* relative retrieval as equivalence relation

Candidate Set

- ☐ RRRAF: ReformulateRelativeRetrievalAsFirst (+ 1 ReformAsExtreme)
- ☐ RRRAL: ReformulateRelativeRetrievalAsLast (+ 1 ReformAsExtreme) (+ 1 ReformUnnecessary) (+ 2 *ReformulateRelativeRetrievalAsLast)

➤ *General Rules*: ReformAsExtreme

➤ *Method Specific Rules*: *ReformulateRelativeRetrievalAsLast

➤ *Resource Rules*: *ReformUnnecessary

Method Ordering: RRRAL(+ 4), RRRAF(+ 1)

Step 1.15: *Equivalence* last(peas@p) and p

Candidate Set

- ☐ A1: Anchor1
- ☐ A2: Anchor2 (+ 2 *Anchor2a)

➤ *Method Specific Rules*: *Anchor2a

Method Ordering: Anchor2(+ 2), Anchor1(-)

Step 1.16: *Reformulate* last(peas@p) as p

Candidate Set

- ☐ RAO: ReformulateAsObject (+ 1 ReformUnnecessary) (+ 1 ReadyToGo)

➤ *Resource Rules*: ReformUnnecessary, ReadyToGo

Method Ordering: RAO(+ 2)

Step 1.17: *Isolate* last(peas)

Candidate Set

- ☐ FGIR: FoldGenericIntoRelation (+ 2 *FGIR)

➤ *Method Specific Rules*: *FoldGenericIntoRelation

Method Ordering: FGIR(+ 3)

Step 1.18: *MaintainIncrementally* last_package

Candidate Set

- ☐ SMFDR: ScatterMaintenanceForDerivedRelation (+ 2 *SMFDR)

➤ *Method Specific Rules*: *ScatterMaintenanceForDerivedRelation

Method Ordering: SMFDR(+2)

Step 1.19: *Remove* reference peas from spec

Candidate Set

☐ BabyWithBathWater

- BWBW1: Y bound to ***concat*** (-2 *BabyWithBathWater3)
- BWBW2: Y bound to ***last*** (-2 *BabyWithBathWater3)
- BWBW3: Y bound to ***update*** (-2 *BabyWithBathWater3)
- BWBW4: Y bound to ***atomic*** (-2 *BabyWithBathWater3)
- BWBW5: Y bound to ***demon*** (-1 *BabyWithBathWater2)

☐ MegaMove (+1 FillIn) (< RemoveRef4)

- MM1: Y bound to ***concat*** (+2 *MegaMove1) (< RemoveRef2) (> RemoveRef1)
- MM2: Y bound to ***last*** (+2 *MegaMove1) (> RemoveRef2) (> RemoveRef1)

☐ PositionalMegaMove (+1 FillIn) (< RemoveRef4) (< RemoveRef1)

- PMM1: Y bound to ***concat*** (+1 *PositionalMegaMove) (< RemoveRef3)
- PMM2: Y bound to ***last*** (+1 *PositionalMegaMove) (+1 ReformUnnecessary) (> RemoveRef3)

☐ RemoveByObjectizingContext (+1 FillIn)

- RBOC1: Y bound to ***concat***
- RBOC2: Y bound to ***last*** (+2 *RemoveByObjectizingContext) (> RemoveRef4)

☐ ReplaceRefWithValue (+1 FillIn) (-2 *ReplaceRefWithValue)

➤ *General Rules*: FillIn

➤ *Method Specific Rules*: *RemoveByObjectizingContext, *MegaMove1, *BabyWithBathWater, *PositionalMegaMove

➤ *Resource Rules*: ReformUnnecessary

➤ *Ordering Rules*: RemoveRef1, RemoveRef2, RemoveRef3, RemoveRef4

Method Ordering: RBOC2(+3), MM2(+3), MM1(+3), PMM2(+3), PMM1(+2), RBOC1(+1)

Step 1.20: *Reformulate* last(peas@p) as object

Candidate Set

☐ RAO: ReformulateAsObject (+1 ReformUnnecessary) (+1 ReadyToGo)

➤ *Resource Rules*: ReformUnnecessary, ReadyToGo

Method Ordering: RAO(+2)

Step 1.21: *Remove* update peas from spec

Candidate Set

☐ BabyWithBathWater

- BWBW1: Y bound to ***atomic*** (-2 *BabyWithBathWater3)
- BWBW2: Y bound to ***demon*** (-1 *BabyWithBathWater2)

☐ RUA: RemoveUnusedAction (+2 *RemoveUnusedAction1)isel()

➤ *Method Specific Rules*: *RemoveUnusedAction1

Method Ordering: RUA(+2)

Step 1.22: *Show* update unnoticed

Candidate Set

☐ SD: ShowDysteleological (+1 *ReadyToGo) (+2 *ShowDysteleological)

➤ *Method Specific Rules*: *ShowDysteleological

➤ *Resource Rules*: ReadyToGo

Method Ordering: SD(+3)

D.2. Remove PREVIOUS_PACKAGE

Step 2.1: *Remove* previous_package

Candidate Set

☐ RR: RemoveRelation (+2 BurnedOutHulk) (+2 *RemoveRelation2)

➤ *General Rules*: BurnedOutHulk

➤ *Method Specific Rules*: *RemoveRelation2

Method Ordering: RR(+4)

Step 2.2: *Remove* reference previous_package from spec

Candidate Set

☐ BabyWithBathWater

- BWBW1: Y bound to ***conditional*** (0 *BabyWithBathWater1)
- BWBW2: Y bound to ***demon*** (-1 *BabyWithBathWater2)

☐ MegaMove (+2 FillIn) (< RemoveRef6)

- MM: Y bound to ***attribute-reference*** (+2 *MegaMove1)

☐ PositionalMegaMove (+1 FillIn) (< RemoveRef6)

- PMM: Y bound to ***attribute-reference*** (+1 *PositionalMegaMove)

☐ RemoveByObjectizingContext (+1 FillIn)

- RBOC: Y bound to ***attribute-reference***

☐ RRWV: ReplaceRefWithValue (+1 FillIn) (+2 *ReplaceRefWithValue1)(> RemoveRef6)

➤ *General Rules*: FillIn

➤ *Method Specific Rules*: *MegaMove1, *BabyWithBathWater, *ReplaceRefWithValue1

➤ *Ordering Rules*: RemoveRef6

Method Ordering: RRWV(+3), MM(+3), PMM(+2), RBOC(+1)

Step 2.3: *Show* value known of previous_package

Candidate Set

☐ ShowUpdateGivesValue

- SUGV: U bound to ***update*** in notice_new_package_at_source (+2 *ShowUpdateGivesValue)

➤ *Method Specific Rules*: *ShowUpdateGivesValue

__Method Ordering__: SUGV(+2)

Step 2.4: *Show* last_package still holds at conditional

__Candidate Set__

☐ SNVSV: ShowNewValueStillValid (+2 *ShowNewValueStillValid)isel()

__Method Ordering__: SNVSV(+2)

Step 2.5: *Show* last_package doesn't change

__Candidate Set__

☐ MoveInterveningUpdate

* MIU: L bound to ***update*** in notice_new_package_at_source (+1 ReadyToGo) (+2 *MoveInterveningUpdate)isel()

➤ *Method Specific Rules*: *MoveInterveningUpdate

➤ *Resource Rules*: ReadyToGo

__Method Ordering__: MIU(+3)

Step 2.6: *ComuteSequentially* conditional before update of last_package

__Candidate Set__

☐ MOOA: MoveOutOfAtomic (+2 *MoveOutOfAtomic)

➤ *Method Specific Rules*: *MoveOutOfAtomic

__Method Ordering__: MOOA(+2)

Step 2.7: *Unfold* atomic

__Candidate Set__

☐ UA: UnfoldAtomic (+5 *UnfoldAtomic)

➤ *Method Specific Rules*: *UnfoldAtomic

__Method Ordering__: UA(+5)

Comment: ***A weight of +5 implies that there is no other method, now or foreseen, which can achieve the goal. In some sense, the goal is an abstract pointer to the method.***

Step 2.8:(*reposted*) *ComuteSequentially* conditinal before update of last_package

__Candidate Set__

☐ CTMS: ConsolidateToMakeSequential (+2 *ConsolidateToMakeSequential)

> *Method Specific Rules*: *ConsolidateToMakeSequential

Method Ordering: CTMS(+2)

Step 2.9: *Consolidate* notice_new_package_at_source and release_package_into_network

Candidate Set

☐ MD: MergeDemons (+5 *MergeDemons)

> *Method Specific Rules*: *MergeDemons

Method Ordering: MD(+5)

> *Action Ordering Rules*: TriggersAlmostEquiv

Step 2.10: *Equivalence* declaration lists

Candidate Set

☐ A1: Anchor1

☐ A2: Anchor2

☐ ECS: EquivalenceCompoundStructures2 (+2 *EquivalenceCompoundStructures2)

> *Method Specific Rules*: *EquivalenceCompoundStructures2

Method Ordering: ECS(+2)

Step 2.11: *Equivalence* p and p.new

Candidate Set

☐ A1: Anchor1 (+2 *Anchor1a) (< EquivVars1)

☐ A2: Anchor2 (+2 *Anchor2a) (> EquivVars1)

> *Method Specific Rules*: *Anchor1a, *Anchor2a

> *Ordering Rules*: EquivVars1

Method Ordering: A2(+2)

Comment: *Until have theory of mnemonics, user relied upon to select names.*

Step 2.12: *Reformulate* p as p.new

Candidate Set

☐ RV: RenameVar (+ 2 *RenameVar)

➤ *Method Specific Rules*: *RenameVar

Method Ordering: RV(+ 2)

Step 2.13:(*reposted*) *ComuteSequentially* conditional before update of last_package

Candidate Set

☐ SU: SwapUp (+ 2 *SwapUp)

➤ *Method Specific Rules*: *SwapUp

Method Ordering: SU(+ 2)

Step 2.14: *Swap* update of last_package with conditional

Candidate Set

☐ SS: SwapStatements (+ 5 *SwapStatements)

➤ *Method Specific Rules*: *SwapStatements

Method Ordering: SS(+ 5)

D.3. Remove LAST_PACKAGE

Step 3.1:*(user)* Remove last_package

Candidate Set

☐ RR: RemoveRelation (+2 BurnedOutHulk) (+2 *RemoveRelation3)

➤ *General Rules*: BurnedOutHulk

➤ *Method Specific Rules*: *RemoveRelation3

Method Ordering: RR(+4)

Step 3.2: *Remove* reference last_package from spec

Candidate Set

☐ BabyWithBathWater

- BWBW1: Y bound to ***conditional*** (0 *BabyWithBathWater1)
- BWBW2: Y bound to ***demon*** (-1 *BabyWithBathWater2)

☐ MegaMove (+1 FillIn)

- MM: Y bound to ***attribute-reference*** (+2 *MegaMove1) (> RemoveRef1)

☐ PositionalMegaMove (+1 FillIn) (< RemoveRef1)

- PMM: Y bound to ***attribute-reference*** (+1 *PositionalMegaMove)

☐ RemoveByObjectizingContext

- RBOC: Y bound to ***attribute-reference***

☐ RRWV: ReplaceRefWithValue

➤ *General Rules*: FillIn

➤ *Method Specific Rules*: *MegaMove1, *BabyWithBathWater, *PositionalMegaMove

➤ *Ordering Rules*: RemoveRef1

Method Ordering: MM(+3), PMM(+2), {RBOC(-), RRWV(-)}

Step 3.3: *Isolate* last_package:destination

Candidate Set

☐ FGIR: FoldGenericIntoRelation (+5 *FoldGenericIntoRelation)

➤ *Method Specific Rules*: *FoldGenericIntoRelation

Method Ordering: FGIR(+5)

Step 3.4: *MaintainIncrementally* last_package_destination

Candidate Set

☐ SMFDR: ScatterMaintenanceForDerivedRelation (+ 2 ScatterMaintenanceForDerivedRelation)

➤ *Method Specific Rules*: •ScatterMaintenanceForDerivedRelation

Method Ordering: SMFDR(+2)

Step 3.5: *Remove* update of last_package

Candidate Set

☐ BabyWithBathWater

• BWBW1: Y bound to ***atomic*** (-2 •BabyWithBathWater3)

• BWBW2: Y bound to ***demon*** (-1 •BabyWithBathWater2)

☐ RUA: RemoveUnusedAction (+ 2 •RemoveUnusedAction1)

➤ *Method Specific Rules*: •BabyWithBathWater2, •BabyWithBathWater3, •RemoveUnusedAction

Method Ordering: RUA(+2)

D.4. Map DID_NOT_SET_SWITCH_WHEN_HAD_CHANCE

Step 4.1: *(user)* *Map* did_not_set_switch_when_had_chance

Candidate Set

- ☐ MCAD: MapConstraintAsDemon (+1 DemonsAreGood) (+2 *MCAD)
- ☐ UC: UnfoldConstraint

➤ *General Rules*: DemonsAreGood

➤ *Method Specific Rules*: *MCAD

Method Ordering: MCAD(+3)

Comment: *Of course the difficult decision here is determining whether a pridictive or backtracking solution is possible. The system points out the need for making the decision, the user provides the answer.*

Step 4.2: *Show* body implies Q

Candidate Set

- ☐ ConjunctImpliesConjunctArm (+1 UseConjunctArm)
  - CICA1: A bound to first conjunct arm (-2 *CICA2)
  - CICA2: A bound to second conjunct arm (-2 *CICA2)
  - CICA3: A bound to third conjunct arm (+2 *CICA1)

➤ *General Rules*: UseConjunctArm

➤ *Method Specific Rules*: *ConjunctImpliesConjunctArm1, *ConjunctImpliesConjunctArm2

Method Ordering: CICA3(+3)

Comment: *The system points out the selection conditions which must be attended to; the user determines which of the candidates satisfies the conditions.*

Step 4.3: *Map* set_switch_when_have_chance (sswhc)

Candidate Set

- ☐ CD: CasifyDemon (+2 CasifyComplexConstruct) (< MapDemon1)
- ☐ MapByConsolidation
  - MBC1: D2 boudn to set_switch (+2 *MBC2) (> MapDemon1)
  - MBC2: D2 bound to release_package_into_network (+1 *MBC1)
  - MBC3: D2 bound to misrouted_package_reached_bin

- *MBC4: D2 bound to create_package (+2 *MBC2) (-2 *MBC4)
- *MBC5: D2 bound to move_package (+2 *MBC2) (-2 *MBC4)
- *MBC6: D2 bound to package_entering_sensor (+1 *MBC1)
- *MBC7: D2 bound to package_leaving_sensor (+1 *MBC1)

☐ UD: UnfoldDemon (+2 *UD) (< MapDemon1)

➤ *General Rules*: CasifyComplexConstruct

➤ *Method Specific Rules*: *MapByConsolidation1, *MapByConsolidation2, *MapByConsolidation4, *UnfoldDemon

➤ *Ordering Rules*: MapDemon1

Method Ordering: MBC1(+2), {CD(+2), UD(+2)}, <MBC2(+1), MBC6(+1), MBC7(+1)}

Step 4.4: *Consolidate* sswhc and set_switch

Candidate Set

☐ MD: MergeDemons (+5 *MergeDemons)

➤ *Method Specific Rules*: *MergeDemons

Method Ordering: MD(+5)

Step 4.5: *Equivalence* two triggers

Candidate Set

☐ A1: Anchor1

☐ A2: Anchor2 (+5 *Anchor2b)

➤ *Method Specific Rules*: *Anchor2b

Method Ordering: A2(+5)

Step 4.6: *Reformulate* random as specific

Candidate Set

☐ SR: SpecializeRandom (+5 *SpecializeRandom)

➤ *Method Specific Rules*: *SpecializeRandom

Method Ordering: SR(+5)

Step 4.7: *(user)* Map require ~P from ThisEvent until EverMore

<u>Candidate Set</u>

- ☐ CPC: CasifyPosConstraint (+2 CasifyComplexConstruct) (> MapConstraint1)
- ☐ MCTA: MoveConstraintToAction
- ☐ NXUX: NotXUntilX
- ☐ TIC: TriggerImpliesConstraint
- ☐ UC: UnfoldConstraint (+2 *UnfoldConstraint) (< MapConstraint1)

➤ *General Rules*: CasifyComplexConstruct

➤ *Method Specific Rules*: *UnfoldConstraint

➤ *Ordering Rules*: MapConstraint1

<u>Method Ordering</u>: CPC(+2), UC(+2), {MCTA(-), NXUX(-), TIC(-)}

Step 4.8: *Casify* require ~P from ThisEvent until EverMore

<u>Candidate Set</u>

- ☐ BS: BinarySplit (+1 ReadyToGo) (-2 *BinarySplit2)
- ☐ PI: PastInduction
- ☐ CFUEC: CasifyFromUntilEverConstraint (+1 ReformUnnecessary) (+1 RequireReformUnnecessary)
- ☐ CAE: CasifyAroundEvent

➤ *Method Specific Rules*: *BinarySplit2

➤ *Resource Rules*: ReformUnnecessary, RequireReformUnnecessary, ReadyToGo

<u>Method Ordering</u>: CFUEC(+2), {PI(-), CAE(-)}

Step 4.9: *Map* require ~P at ThisEvent

<u>Candidate Set</u>

- ☐ CPC: CasifyPosConstraint (+2 CasifyComplexStructure) (> MapConstraint1) (< MapConstraint2)
- ☐ MCAC: MoveConstraintToAction
- ☐ NXUX: NotXUntilX
- ☐ TIC: TriggerImpliesConstraint (+1 ReformUnnecessary) (+1 RequireReformUnnecessary) (+1 ReadyToGo) (> MapConstraint2)
- ☐ UC: UnfoldConstraint (+2 *UnfoldConstraint) (< MapConstraint1) (< MapConstraint2)

➤ *General Rules*: CasifyComplexConstruct

➤ *Method Specific Rules*: *UnfoldConstraint

➤ *Resource Rules*: ReadyToGo, ReformUnnecessary, RequireReformUnnecessary

➤ *Ordering Rules*: MapConstraint1, MapConstraint2

Method Ordering: TIC(+3), CPC(+2), UC(+2)

Step 4.10: *Map* require ~P after ThisEvent

Candidate Set

- ☐ CPC: CasifyPosConstraint (+2 CasifyComplexConstruct) (> MapConstraint1)
- ☐ MCTA: MoveConstraintToAction
- ☐ NXUX: NotXUntilX
- ☐ TIC: TriggerImpliesConstraint
- ☐ UC: UnfoldConstraint (+2 *UC) (< MapConstraint1)

➤ *General Rules*: CasifyComplexConstruct

➤ *Method Specific Rules*: *UnfoldConstraint

➤ *Ordering Rules*: MapConstraint1

Method Ordering: CasifyPosConstraint(+2), UnfoldConstraint(+2)

Step 4.11: *Casify* require ~P after ThisEvent

Candidate Set

- ☐ BinarySplit (+1 ReadyToGo) (-2 *BinarySplit2)
- ☐ PastInduction
- ☐ CasifyFromUntilEverConstraint
- ☐ CasifyAroundEvent (+1 ReformUnnecessary) (+1 RequireReformUnnecessary)

➤ *Method Specific Rules*: *BinarySplit2

➤ *Resource Rules*: ReadyToGo, ReformUnnecessary, RequirteReformUnnecessary

Method Ordering: CasifyAroundEvent(+2), {PastInduction(-), CasifyFromUntilEverConstraint(-)}

Step 4.12: *Map* require ~P after ThisEvent until E

Candidate Set

- ☐ CasifyPosConstraint (+2 CasifyComplexStructure) (> MapConstraint1) (< MapConstraint2)
- ☐ MoveConstraintToAction

☐ NotXUntilX (+1 ReformUnnecessary) (+1 RequireReformUnnecessary) (> MapConstraint2)

☐ TriggerImpliesConstraint

☐ UnfoldConstraint (+2 *UC) (< MapConstraint1) (< MapConstraint2)

➤ *General Rules*: CasifyComplexConstruct

➤ *Method Specific Rules*: ReadyToGo, ReformUnnecessary, RequireReformUnnecessary

➤ *Ordering Rules*: MapConstraint1, MapConstraint2

Method Ordering: NotXUntilX(+2), CasifyPosConstraint(+2), UnfoldConstraint(+2)

Step 4.13: *Map* ~P during E

Candidate Set

☐ CasifyPosConstraint (+2 CasifyComplexStructure) (> MapConstraint1)

☐ MoveConstraintToAction

☐ NotXUntilX

☐ TriggerImpliesConstraint

☐ UnfoldConstraint (+2 *UnofldConstraint) (< MapConstraint1)

➤ *General Rules*: CasifyComplexConstruct

➤ *Method Specific Rules*: *UnfoldConstraint

➤ *Ordering Rules*: MapConstraint1

Method Ordering: CasifyPosConstraint(+2), UnfoldConstraint(+2), {MoveConstraintToAction(-), NotXUntilX(-), TriggerImpliesConstraint(-)}

Step 4.14: *Casify* require ~P during E

Candidate Set

☐ BinarySplit (+1 ReadyToGo) (-2 *BinarySplit2)

☐ PastInduction (+1 ReformUnnecessary) (+1 RequireReformUnnecessary)

☐ CasifyFromUntilEverConstraint

☐ CasifyAroundEvent

➤ *Method Specific Rules*: *BinarySplit2

➤ *Resource Rules*: ReadyToGo, ReformUnnecessary, RequireReformUnnecessary

Method Ordering: PastInduction(+2), {CasifyFromUntilEverConstraint(-), CasifyAroundEvent(-)}

Step 4.15: *Map* require ~P at last update switch_setting

<u>Candidate Set</u>

- ☐ CasifyPosConstraint (+2 CasifyComplexStructure) (> MapConstraint1) (< MapConstraint3)
- ☐ MoveConstraintToAction (+1 ReformUnnecessary) (+1 RequireReformUnnecessary) (> MapConstraint3)
- ☐ NotXUntilX
- ☐ TriggerImpliesConstraint
- ☐ UnfoldConstraint (+2 *UnfoldConstraint) (< MapConstraint1)

➤ *General Rules*: CasifyComplexConstruct

➤ *Method Specific Rules*: *UnfoldConstraint

➤ *Resource Rules*: ReformUnnecessary, RequireReformUnnecessary

➤ *Ordering Rules*: MapConstraint1, MapConstraint3

<u>Method Ordering</u>: MoveConstraintToAction(+2), CasifyPosConstraint(+2), UnfoldConstraint(+2), {NotXUntilX(-), TriggerImpliesConstraint(-)}

Step 4.16: *Map* require ~(<u>start of</u> ~P) between last update, E

<u>Candidate Set</u>

- ☐ CasifyPosConstraint (+2 CasifyComplexStructure) (> MapConstraint1) (< MapConstraint2)
- ☐ MoveConstraintToAction
- ☐ NotXUntilX
- ☐ ShowNoChange (+2 *ShowNoChange) (> MapConstraint2)
- ☐ TriggerImpliesConstraint
- ☐ UnfoldConstraint (+2 *UnfoldConstraint) (< MapConstraint1)

➤ *General Rules*: CasifyComplexConstruct

➤ *Method Specific Rules*: *ShowNoChange

➤ *Ordering Rules*: MapConstraint1, MapConstraint2

<u>Method Ordering</u>: ShowNoChange(+2), CasifyPosConstraint(+2), UnfoldConstraint(+2)

Step 4.17: *Show* ~(start ~P) between last update, E

<u>Candidate Set</u>

- ☐ Ø

Step 4.18: *(user)* *Map* update of switch_setting where P

Candidate Set

- ☐ CNV: ComputeNewValue (+2 *ComputeNewValue)

➤ *Method Specific Rules*: *ComputeNewValue

Method Ordering: CNV(+2)

Step 4.19: *Unfold* switch_set_wrong_for_package at set_switch

Candidate Set

- ☐ SCODR: ScatterComputationOfDerivedRelation (+5 *ScatterComputationOfDerivedRelation)

➤ *Method Specific Rules*: *ScatterComputationOfDerivedRelation

Method Ordering: SCODR(+5)

D.5. Map PACKAGES_DUE_AT_SWITCH

Step 5.1:(*user*) *Map* packages_due_at_switch (pdas)

Candidate Set

☐ MDR: MaintainDerivedRelation (+ 2 *MaintainDerivedRelation) (> MapDR2a)

☐ UDR: UnfoldDerivedRelation (+ 2 *UnfoldDerivedRelation1) (< MapDR2a)

➤ *Method Specific Rules*: *MaintainDerivedRelation, *UnfoldDerivedRelation1

➤ *Ordering Rules*: MapDR2a

Method Ordering: MDR(+ 2), UDR(+ 2)

Comment: *Currently, the system has no mechanism for computing the lefthandside of MapDR2, i.e. it is up to the user to determine the cost of computing the relation.*

Step 5.2: *MaintainIncrementally* pdas

Candidate Set

☐ IntroduceSeqMaintenanceDemon (+ 1 DemonsAreGood) (+ 1 *IntroduceSeqMaintenanceDemon) (+ 1 ReformUnnecessary) (< MaintDR1)

☐ ScatterMaintenanceForDerivedRelation (+ 2 *SMFDR) (> MaintDR1)

➤ *General Rules*: DemonsAreGood

➤ *Method Specific Rules*: *IntroduceSeqMaintenanceDmeon, *ScatterMaintenacneForDerivedRelation

➤ *Resource Rules*: ReformUnnecessary

➤ *Ordering Rules*: MaintDR1

Method Ordering: SMFDR(+ 2), ISMD(+ 3)

Step 5.3: *Flatten* pdas

Candidate Set

☐ Flatten (+ 2 *Flatten)

➤ *Method Specific Rules*: *Flatten

Method Ordering: Flatten(+ 2)

Step 5.4: *Map* location_on_route_to_bin

Candidate Set

☐ StoreExplicitly (+ 2 *StoreExplicitly) (> MapDR1a)

☐ UnfoldDerivedRelation (-2 *UnfoldDerivedRelation2) (< MapDR1a)

➤ *Method Specific Rules*: *StoreExplicitly, *UnfoldDerivedRelation2

➤ *Ordering Rules*: MapDR1a

Method Ordering: StoreExplicitly(+2)

Step 5.5: *Map* misrouted

Candidate Set

☐ MDR: MaintainDerivedRelation (+2 *MaintainDerivedRelation) (< MapDR2b)

☐ UDR: UnfoldDerivedRelation (+2 *UnfoldDerivedRelation1) (> MapDR2b)

➤ *Method Specific Rules*: *MaintainDerivedRelation, *UnfoldDerivedRelation1

➤ *Ordering Rules*: MapDR2b

Method Ordering: MDR(+2), UDR(+2)

Step 5.6: *Unfold* misrouted at pdas

Candidate Set

☐ SCODR: ScatterComputationOfDerivedRelation (+5 *ScatterComputationOfDerivedRelation)

➤ *Method Specific Rules*: *ScatterComputationOfDerivedRelation

Method Ordering: SCODR(+5)

Step 5.7: *Flatten* pdas

Candidate Set

☐ Flatten (+2 *Flatten)

➤ *Method Specific Rules*: *Flatten

Method Ordering: Flatten(+2)

Step 5.8: *Map* switch_set_wrong_for_package

Candidate Set

☐ MDR: MaintainDerivedRelation (+2 *MaintainDerivedRelation) (< MapDR2b)

☐ UDR: UnfoldDerivedRelation (+2 *UnfoldDerivedRelation1) (> MapDR2b)

➤ *Method Specific Rules*: *MaintainDerivedRelation, *UnfoldDerivedRelation1

➤ *Ordering Rules*: MapDR2b

Method Ordering: UDR(+2), MDR(+2)

Step 5.9: *Unfold* switch_set_wrong_for_package

Candidate Set

☐ SCODR: ScatterComputationOfDerivedRelation (+5 *ScatterComputationOfDerivedRelation)

➤ *Method Specific Rules*: *ScatterComputationOfDerivedRelation

Method Ordering: SCODR(+5)

Step 5.10: *Purify* loop in create_package

Candidate Set

☐ PurifyDemon (+2 *PurifyDemon)

➤ *Method Specific Rules*: *PurifyDemon

Method Ordering: PurifyDemon(+2)

Step 5.11: *Remove* loop from create_package

Candidate Set

☐ BabyWithBathWater

- BWBW1: Y bound to ***atomic*** (-2 *BabyWithBathWater3)
- BWBW2: Y bound to ***demon*** (-2 *BabyWithBathWater3)

☐ RFD: RemoveFromDemon (+2 *RemoveFromDemon) (< RemAct1)

☐ RUA: RemoveUnusedAction (+2 *RemoveUnusedAction2) (> RemAct1)

➤ *Method Specific Rules*: *BabyWithBathWater3, *RemoveFromDemon, *RemoveUnusedAction2

➤ *Ordering Rules*: RemAct1

Method Ordering: RUA(+2), RFD(+2)

Comment: *The system does not have the necessary knowledge to determine what code can be simplified away and what must remain. Because of the big gain in problem solving costs, the system always suggests blowing away unfolded code before moving it about. Here, the introduced loop is necessary and hence must be removed from the demon.*

Step 5.12: *Globalize* loop in create_package

Candidate Set

☐ GlobalizeAction (+2 *GlobalizeAction)

➤ *Method Specific Rules*: *GlobalizeAction

Method Ordering: GlobalizeAction(+2)

Step 5.13: *Unfold* atomic

__Candidate Set__

□ UnfoldAtomic (+5 *UnfoldAtomic)

➤ *Method Specific Rules*: *UnfoldAtomic

__Method Ordering__: UnfoldAtomic(+5)

Step 5.14: *Purify* conditional in move_package

__Candidate Set__

□ PurifyDemon (+2 *PurifyDemon)

➤ *Method Specific Rules*: *PurifyDemon

__Method Ordering__: PurifyDemon(+2)

Step 5.15: *Remove* conditional in move_package

__Candidate Set__

□ BabyWithBathWater

- Y bound to ***atomic*** (-2 *BabyWithBathWater3)
- Y bound to ***demon*** (-2 *BabyWithBathWater3)

□ RemoveFromDemon (+2 *RemoveFromDemon) (< RemAct2)

□ RemoveUnusedAction (+2 *RemoveUnusedAction2) (> RemAct1)

➤ *Method Specific Rules*: *BabyWithBathWater3, *RemoveUnusedAction2, *RemoveFromDemon

➤ *Ordering Rules*: RemAct1

__Method Ordering__: RUA(+2), RFD(+2)

Comment: *See comments at 5.11*

Step 5.16: *Globalize* conditional in move_package

__Candidate Set__

□ GlobalizeAction (+2 *GlobalizeAction)

➤ *Method Specific Rules*: *GlobalizeAction

__Method Ordering__: GlobalizeAction(+2)

Step 5.17: *Unfold* atomic

Candidate Set

☐ UnfoldAtomic (+5 *UnfoldAtomic)

➤ *Method Specific Rules*: *UnfoldAtomic

Method Ordering: **UnfoldAtomic(+5)**

Step 5.18: *Casify* package_leaving_sensor

Candidate Set

☐ CasifySuperTrigger (+2 *CasifySuperTrigger)

➤ *Method Specific Rules*: *CasifySuperTrigger

Method Ordering: **CasifySuperTrigger(+2)**

Step 5.19: *Casify* package_entering_sensor

Candidate Set

☐ CasifySuperTrigger (+2 *CasifySuperTrigger)

➤ *Method Specific Rules*: *CasifySuperTrigger

Method Ordering: **CasifySuperTrigger(+2)**

D.6. Map Demons

Step 6.1:*(user)* *Map* set_switch

Candidate Set

☐ CD: CasifyDemon (+2 CasifyComplexConstruct) (+2 *CasifyDemon)

☐ MapByConsolidation

- MBC1: D2 bound to release_package_into_network (+1 *MBC1)
- MBC:2 D2 bound to package_entering_switch (+1 *MBC1)
- MBC3: D2 bound to package_entering_bin (+1 *MBC1)
- MBC4: D2 bound to package_leaving_switch (+1 *MBC1)
- MBC5: D2 bound to package_leaving_bin (+1 *MBC1)
- MBC6: D2 bound to init_memo (+1 *MBC1)
- MBC7: D2 bound to misrouted_package_reached_bin
- MBC8: D2 bound to create_package (-2 *MBC4) (+1 *MBC2)
- MBC9: D2 bound to move_package (-2 *MBC4) (+1 *MBC2)

☐ UD: UnfoldDemon (+1 *UnfoldDemon)

➤ *General Rules*: CasifyComplexConstruct

➤ *Method Specific Rules*: *CasifyDemon, *MBC1, *MBC2, *MBC4, *UnfoldDemon

Method Ordering: CD(+4), {MBC1(+1), MBC2(+1), MBC3(+1), MBC4(+1), MBC5(+1), MBC6(+1), UD(+1)}

Step 6.2: *Casify* set_switch

Candidate Set

☐ CCT: CasifyConjunctiveTrigger (+2 *CasifyConjunctiveTrigger)

➤ *Method Specific Rules*: *CasifyConjunctiveTrigger

Method Ordering: CCT(+2)

Step 6.3: *Map* set_switch_when_bubble_package (sswbp)

Candidate Set

☐ CD: CasifyDemon

☐ MapByConsolidation

- MBC1: D2 bound to release_package_into_network (+1 *MBC1)
- MBC:2 D2 bound to package_entering_switch (+1 *MBC1)
- MBC3: D2 bound to package_entering_bin (+1 *MBC1)
- MBC4: D2 bound to package_leaving_switch (+1 *MBC1)
- MBC5: D2 bound to package_leaving_bin (+1 *MBC1)
- MBC6: D2 bound to init_memo (+1 *MBC1)
- MBC7: D2 bound to misrouted_package_reached_bin
- MBC8: D2 bound to set_switch_on_exit (+1 *MBC1) (-2 *MBC5)
- MBC9: D2 bound to create_package (-2 *MBC4) (+1 *MBC2)
- MBC10: D2 bound to move_package (-2 *MBC4) (+1 *MBC2)

☐ UD: UnfoldDemon (+1 *UnfoldDemon)

➤ *Method Specific Rules*: *MBC1, *MBC2, *MBC4, *MBC5, *UnfoldDemon

Method Ordering: {MBC1(+1), MBC2(+1), MBC3(+1), MBC4(+1), MBC5(+1), MBC6(+1), UD(+1)}

Comment: *User determines that consolidation doesn't look promising. Unfolding a demon is a strategy that in general always works. It is often not a great choice because of the necessary work of opotimizing the unfolded code. Here it is about the only choice.*

Step 6.4: *Unfold* sswbp at release_package_into_network

Candidate Set

☐ ScatterComputationOfDemon (+5 *ScatterComputationOfDemon)

➤ *Method Specific Rules*: *ScatterComputationOfDemon

Method Ordering: ScatterComputationOfDemon(+5)

Step 6.5: *Factor* update of packages_due_at_switch

Candidate Set

☐ FactorDBMaintenanceIntoAction (+1 ReadyToGo) (+2 *FactorDBMaintenanceIntoAction)

➤ *Method Specific Rules*: *FactorDBMaintenanceIntoAction

➤ *Resource Rules*: ReadyToGo

Method Ordering: FactorDBMaintenanceIntoAction(+3)

Step 6.6: *Map* set_switch_on_exit

Candidate Set

- ☐ CD: CasifyDemon
- ☐ MapByConsolidation
  - * MBC1: D2 bound to release_package_into_network (+1 *MBC1)
  - * MBC:2 D2 bound to package_entering_switch (+1 *MBC1)
  - * MBC3: D2 bound to package_entering_bin (+1 *MBC1)
  - * MBC4: D2 bound to package_leaving_switch (+1 *MBC1)
  - * MBC5: D2 bound to package_leaving_bin (+1 *MBC1)
  - * MBC6: D2 bound to init_memo (+1 *MBC1)
  - * MBC7: D2 bound to misrouted_package_reached_bin
  - * MBC8: D2 bound to create_package (-2 *MBC4) (+1 *MBC2)
  - * MBC9: D2 bound to move_package (-2 *MBC4) (+1 *MBC2)
- ☐ UD: UnfoldDemon (+1 *UnfoldDemon)

➤ *Method Specific Rules*: *MBC1, *MBC2, *MBC4, *UnfoldDemon

Method Ordering: {MBC1(+1), MBC2(+1), MBC3(+1), MBC4(+1), MBC5(+1), MBC6(+1), UD(+1)}

Comment: *Again up to the user to find a promising consolidation demon. In this case, a level of indirection is involved vis a vis the derived relation* SWITCH_IS_EMPTY.

Step 6.7: *Consolidate* set_switch_on_exit and package_leaving_switch

Candidate Set

- ☐ MergeDemons (+5 *MergeDemons)

➤ *Method Specific Rules*: *MergeDemons

Method Ordering: MergeDemons(+5)

Step 6.8: *Equivalence* triggers

Candidate Set

- ☐ Anchor1 (+2 *Anchor1c)
- ☐ Anchor2

➤ *Method Specific Rules*: *Anchor1c

Method Ordering: Anchor1(+2), Anchor2(-)

Comment: *Note that the selection rule *Anchor1c focuses the user's*

attention in the right place, the body of **SWITCH_IS_EMPTY**. *Currently, the user is required to carry on from here in regards to the evaluation of promising.*

Step 6.9: *Reformulate* switch_is_empty as expression

Candidate Set

☐ ReformulateDerivedRelation (+2 *ReformulateDerivedRelation)

➤ *Method Specific Rules*: *ReformulateDerivedRelation

Method Ordering: ReformulateDerivedRelation(+2)

Step 6.10: *Unfold* switch_is_empty in trigger

Candidate Set

☐ ScatterComputationOfDerivedRelation (+5 *ScatterComputationOfDerivedRelation)

➤ *Method Specific Rules*: *ScatterComputationOfDerivedRelation

Method Ordering: ScatterComputationOfDerivedRelation(+5)

Step 6.11: *Reformulate* existential as universal

Candidate Set

☐ ReformulateExistentialTrigger (+2 *ReformulateExistentialTrigger)

➤ *Method Specific Rules*: *ReformulateExistentialTrigger

Method Ordering: ReformulateExistentialTrigger(+2)

Step 6.12: *Equivalence* two declarations

Candidate Set (Problem Solving Abridgement)

☐ Anchor1 (+2 *Anchor1a) (< EquivVars1)

☐ Anchor2 (+2 *Anchor2a) (> EquivVars1)

➤ *Method Specific Rules*: *Anchor1a, *Anchor2a

➤ *Ordering Rules*: EquivVars1

Method Ordering: Anchor2(+2), Anchor1(+2)

Step 6.13:(*user*) Map misrouted_package_reached_bin

Candidate Set

☐ CD: CasifyDemon (+2 CasifyComplexConstruct) (+2 *CasifyDemon1)

☐ MapByConsolidation

* MBC1: D2 bound to release_package_into_network (+1 *MBC1)
* MBC:2 D2 bound to package_entering_switch (+1 *MBC1)
* MBC3: D2 bound to package_entering_bin (+1 *MBC1)
* MBC4: D2 bound to package_leaving_switch (+1 *MBC1)
* MBC5: D2 bound to package_leaving_bin (+1 *MBC1)
* MBC6: D2 bound to init_memo (+1 *MBC1)
* MBC7: D2 bound to misrouted_package_reached_bin
* MBC8: D2 bound to create_package (-2 *MBC4) (+1 *MBC2)
* MBC9: D2 bound to move_package (-2 *MBC4) (+1 *MBC2)

☐ UD: UnfoldDemon (+1 *UnfoldDemon)

➤ *Method Specific Rules*: *CasifyDemon1, *MBC1, *MBC2, *MBC4, *UnfoldDemon

<u>Method Ordering</u>: CD(+4), {MBC1(+1), MBC2(+1), MBC3(+1), MBC4(+1), MBC5(+1), MBC6(+1), UD(+1)}

Step 6.14: *Casify* misrouted_package_reached_bin

<u>Candidate Set</u>

☐ CasifyConjunctiveTrigger (+2 *CasifyConjunctiveTrigger)

➤ *Method Specific Rules*: *CasifyConjunctiveTrigger

<u>Method Ordering</u>: CasifyConjunctiveTrigger(+2)

Step 6.15: *Map* misrouted_package_located_at_bin

<u>Candidate Set</u>

☐ CD: CasifyDemon

☐ MapByConsolidation

* MBC1: D2 bound to release_package_into_network
* MBC:2 D2 bound to package_entering_switch
* MBC3: D2 bound to package_entering_bin (+2 *MBC8)
* MBC4: D2 bound to package_leaving_switch
* MBC5: D2 bound to package_leaving_bin

* MBC6: D2 bound to init_memo
* MBC7: D2 bound to misrouted_package_reached_bin
* MBC8: D2 bound to create_package (-2 *MBC4) (+1 *MBC2)
* MBC9: D2 bound to move_package (-2 *MBC4) (+1 *MBC2)

☐ UD: UnfoldDemon (+1 *UnfoldDemon)

➤ *Method Specific Rules*: *MBC2, *MBC4, *MBC6, *UnfoldDemon

Method Ordering: MBC3(+2), UD(+1), {MBC1(-), MBC2(-), MBC4(-), MBC5(-), MBC6(-), MBC7(-)}

Step 6.16: *Consolidate* misrouted_package_located_at_bin and

Candidate Set

☐ MergeDemons (+5 *MergeDemons)

➤ *Method Specific Rules*: *MergeDemons

Method Ordering: MergeDemons(+5)

➤ *Action Ordering Rules*: TriggersAlmostEquiv

Step 6.17: *Equivalence* declaration lists

Candidate Set

☐ A1: Anchor1

☐ A2: Anchor2

☐ ECS: EquivalenceCompoundStructures2 (+2 *ECS2)

➤ *Method Specific Rules*: *ECS2

Method Ordering: ECS2(+2)

Step 6.18: *Equivalence* bin.reached and bin

Candidate Set

☐ Anchor1 (+2 *Anchor1a) (> EquivVars1)

☐ Anchor2 (+2 *Anchor2a) (< EquivVars1)

➤ *Method Specific Rules*: *Anchor1a, *Anchor2a

➤ *Ordering Rules*: EquivVars1

Method Ordering: Anchor1(+2), Anchor2(+2)

Step 6.19:(*reposted*) *Equivalence* declaration lists

Candidate Set

- ☐ A1: Anchor1
- ☐ A2: Anchor2
- ☐ ECS: EquivalenceCompoundStructures2
- ☐ ANV: AddNewVar (+2 *AddNewVar)

➤ *Method Specific Rules*: *AddNewVar

Method Ordering: ANV(+2)

Step 6.20: *Map* misrouted_package_destination_set

Candidate Set

- ☐ CD: CasifyDemon
- ☐ MapByConsolidation
  - MBC1: D2 bound to release_package_into_network (+1 *MBC1)
  - MBC:2 D2 bound to package_entering_switch (+1 *MBC1)
  - MBC3: D2 bound to package_entering_bin (+1 *MBC1)
  - MBC4: D2 bound to package_leaving_switch (+1 *MBC1)
  - MBC5: D2 bound to package_leaving_bin (+1 *MBC1)
  - MBC6: D2 bound to init_memo (+1 *MBC1)
  - MBC7: D2 bound to misrouted_package_reached_bin
  - MBC8: D2 bound to create_package (-2 *MBC4) (+1 *MBC2)
  - MBC9: D2 bound to move_package (-2 *MBC4) (+1 *MBC2)
- ☐ UD: UnfoldDemon (+1 *UnfoldDemon)

➤ *Method Specific Rules*: *MBC1, *MBC2, *MBC4, *UnfoldDemon

Method Ordering: {MBC1(+1), MBC2(+1), MBC3(+1), MBC4(+1), MBC5(+1), MBC6(+1), UD(+1)}

Comment: *See 6.3*

Step 6.21: *Unfold* misrouted_package_destination_set

Candidate Set

- ☐ ScatterComputationOfDemon (+5 *SCOD)

> ***Method Specific Rules*:** *SCOD

Method Ordering: SCOD(+5)

Appendix E
Goal Descriptors

In this Appendix, we will present the set of goal descriptors that make up Glitter's development vocabulary. We have attempted to define a *general* set of descriptors, distilling the essential semantics of a development goal and avoiding special cases. For instance, one of the goals of the language is Remove. This goal takes as an argument an arbitrary program structure. We do not define a separate goal for removing particular structures: RemoveRelation, RemoveDemon, etc.

With each descriptor will be given a textual description followed by several examples of the descriptor in use. Heading each example section is a list of the steps in the router development (appendix C) where the goal is *explicitly* used; goals trivially satisfied in the router development (i.e. achieved within the posting state) do not show up explicitly either here or in the development. In some cases, we have taken examples from other developments including the following:

1. Text preprocessor. The first development attempted using Glitter. The problem is the optimization of a procedure which cleans-up a message body before sending it through an analyzer. Portions of the development are reported in [Balzer 76, Wile 81a]. This development will be denoted as *Text Preprocessor*.

2. Line drawing algorithm. This hand development of a graphics line drawing algorithm was reported by Sproull [Sproull 81]. It offers a slightly different view of several development concepts. We will denote this development as *Line Draw*.

3. Heap sort development. No research into automatic program development would be complete without at least one sort example. This one is taken from some unpublished notes of Tim Standish. We will denote this development as *Heap Sort*.

We use these different examples to provide explanation variety; only the Package Router and Text Preprocessor have been developed using Glitter.

Finally, we will simplify the goal posting notation to that used in Appendix B.

E.1. Casify

Casify(C|*construct*)

Achievement Condition: C is replaced with $\{C_1 ... C_n\}$

Goal Description: this is the driver behind divide-and-conquer strategies. A complex structure can often be broken out into several simpler components. However, while the case-analysis concept is a powerful one, the real insight comes from selecting the right partitioning elements. The user is generally relied on to make this selection.

.... **Examples of Use** ....

Router References: 4.8, 4.11, 4.14, 5.18, 5.19, 6.2, 6.14

Example A

Router Reference: 4.11

Development context: section B.4 of the router development points out the problem of working with complex, temporally-modified predicates. At step 4.10, the following constraint is marked for mapping:

```
require (~(package:LOCATED_AT = switch
                and
          SWITCH_SET_WRONG_FOR_PACKAGE(switch,package))
            after ThisEvent
```

In this example, *ThisEvent* can be interpreted as *the current time*. Abstractly, we have

```
require P from now on)
```

Step 4.11 attempts to simplify the mapping problem by suggesting that the single constraint be broken out into several cases. Once the *Casify* goal is posted, the remaining problem is choosing the best case-analysis method. In this example, a method is chosen which casifies around some future event E (chosen by the user):

```
require P from now until E);
require P during E);
require P after E);
```

The time requirement is split into the period before, during and after E. Of course, the effectiveness of casifying here depends on the correct choice of E. In this case E was chosen as the time the package was located at the switch, allowing is to straightforwardly get rid of the first and third cases and center our attention on the second, linchpin requirement.

Example B

Router Reference: 5.18

Development context: while the use of abstraction may lead to a more perspicuous initial spec, the development may require specific cases to be broken out. Such is the case in step 5.18: an abstract (a.k.a. Super) type SENSOR has been defined in the initial spec. Further, a demon has been defined that triggers on a package leaving a sensor.

```
demon PACKAGE_LEAVING_SENSOR(package, sensor)
   trigger ~package:LOCATED_AT = sensor
   response null;
```

In section 5 of the development, it becomes useful to know which type of sensor (SWITCH or BIN) a package is leaving. The case-analysis method chosen hinges on the subtypes of SENSOR, producing two new demons:

```
demon PACKAGE_LEAVING_SWITCH(package, switch)
   trigger ~package:LOCATED_AT = switch
   response null;

demon PACKAGE_LEAVING_BIN(package, bin)
   trigger ~package:LOCATED_AT = bin
   response null;
```

Example C

Router Reference: 6.13

Development context: the triggering of a constraint or demon may depend on the occurrence of any one of a number of events. It is sometimes useful to break out the events into individual cases, and treat each one separately. Such is the case in step 6.13, the mapping of the demon MISROUTED_PACKAGE_REACHED_BIN (note that Gist variable convenetions do not allow *bin.reached* and *bin.intended* to be boudn to the same physicla bin):

```
demon MISROUTED_PACKAGE_REACHED_BIN(package, bin.reached, bin.intended)
  trigger package:LOCATED_AT = bin.reached
               and
          package:DESTINATION = bin.intended
  response invoke MISROUTED_ARRIVAL(bin.reached, bin.intended)
```

The necessary conditions for triggering this demon are either 1) a package enters a bin or b) the destination of a package is set[65]. Breaking the demon into these two cases facilitates further development: the second case cannot be satisfied and hence only the first need be considered (in its now simplified form):

```
demon MISROUTED_PACKAGE_LOCATED_AT_BIN(package,bin.reached,bin-intended)
  trigger package:LOCATED_AT = bin.reached
  response
          if (package:DESTINATION = bin.intended
                     at ThisEvent);
            then invoke MISROUTED_ARRIVAL(bin.reached, bin.intended);

demon MISROUTED_PACKAGE_DESTINATION_SET(package,bin.reached,bin-intended)
  trigger package:DESTINATION = bin.intended
  response
       if (package:LOCATED_AT = bin.reached
                     at ThisEvent);
         then invoke MISROUTED_ARRIVAL(bin.reached, bin.intended);
```

[65]That these two events cannot happen simultaneously is something that must be shown later in the development.

Example D

Router Reference: `Text Preprocessor`

Development context: a portion of the *Text Preprocessor* is given below. The following actions are performed on a sequence of characters *Text*:

- □ ▸$_1$ If the current character is a linefeed then replace it with a space.
- □ ▸$_2$ If the current character is not an alphanumeric or space then remove it from *Text*.
- □ ▸$_3$ If the current character is redundant (i.e. a space preceded by a space) then remove it from *Text*.

```
...
loop Char in Text
   do begin
▸1        if linefeed(Char then invoke REPLACE(Char, space, Text);
▸2        if ~(alphanumeric(Char) or space(Char))
                                  then invoke REMOVE(Char, Text);
▸3        if redundant(Char, Text)
                                  then invoke REMOVE(Char, Text);
   end ...
```

By using the *Casify* goal, we can add some structure which will facilitate further optimization. We can embed the body of the loop within each case of a mutually-exclusive case statement (given that the user supplies the necessary partitioning):

```
...
loop Char in Text do
   mux-case Char
    linefeed: begin
                if linefeed(Char)
                     then invoke REPLACE(Char, space, Text);
                if ~(alphanumeric(Char) or space(Char))
                     then invoke REMOVE(Char, Text);
                if redundant(Char, Text) then invoke REMOVE(Char, Text);
               end
    space: begin
                if linefeed(Char)
                     then invoke REPLACE(Char, space, Text);
                if ~(alphanumeric(Char) or space(Char))
                     then invoke REMOVE(Char, Text);
                if redundant(Char, Text) then invoke REMOVE(Char, Text);
               end
    alphanumeric: begin
                if linefeed(Char)
                     then invoke REPLACE(Char, space, Text);
                if ~(alphanumeric(Char) or space(Char))
                     then invoke REMOVE(Char, Text);
                if redundant(Char, Text) then invoke REMOVE(Char, Text);
               end
    otherwise: begin
                if linefeed(Char)
                     then invoke REPLACE(Char, space, Text);
                if ~(alphanumeric(Char) or space(Char))
                     then invoke REMOVE(Char, Text);
                if redundant(Char, Text) then invoke REMOVE(Char, Text);
               end
   end-mux-case;
...
```

After further optimization, we have

```
...
loop Char in Text do
   mux-case Char
    linefeed: if predecessor(space, Char, Text)
                          then invoke REMOVE(Char, Text)
                          else invoke REPLACE(Char, space, Text);
    space: if predecessor(space, Char, Text)
                           then invoke REMOVE(Char, Text);
    alphanumeric: ;
    otherwise: invoke REMOVE(Char, Text)
   end-mux-case;
...
```

E.2. ComputeSequentially

ComputeSequentially(C1|*construct*, C2|*construct*)

Achievement Condition: C1 computationally precedes C2

Goal Description: C2 is an action that has the potential of effecting C1. We want to guarantee that C2 does not effect C1.

------------------------ **Examples of Use** ------------------------

Router References: 2.6

Example A

Router Reference: 2.6

Development context:

```
demon NOTICE_NEW_PACKAGE_AT_SOURCE(package)
  trigger package:LOCATED_AT = the source
  response
    atomic
▶1      update prev_package in PREVIOUS_PACKAGE($)
             to LAST_PACKAGE(*);
▶2      update last_package in LAST_PACKAGE($)
             to package
    end atomic;

demon RELEASE_PACKAGE_INTO_NETWORK(package.new)
  trigger package.new:LOCATED_AT = the source
  response
    begin
▶3     if PREVIOUS_PACKAGE(*):DESTINATION ≠ package.new:DESTINATION
        then WAIT[];
       update :LOCATED_AT of package.new to (the source):SOURCE_OUTLET
    end;
```

Here, relation PREVIOUS_PACKAGE is updated to LAST_PACKAGE(*). We want to insure that a subsequent reference to PREVIOUS_PACKAGE can be replaced with

LAST_PACKAGE, i.e. that the value of LAST_PACKAGE has not changed between the time PREVIOUS_PACKAGE was updated and the time it is referenced. If there exists an action that changes LAST_PACKAGE between these times, we want the action executed after the reference. Above, ▸$_1$ points to the update of PREVIOUS_PACKAGE, ▸$_2$ points to the change to LAST_PACKAGE which must be moved, and ▸$_3$ to the reference.

Example B

Router Reference: `Text Preprocessor`

During the development of the text-preprocessor, a state is reached containing the following program fragment:

```
...
  begin
▸1     invoke REPLACE(Char newspace Text);
▸2     if predecessor(space, Char, Text))
           then invoke REMOVE(Char Text)
  end
 ...
```

That is, replace the current character *Char* with a space (▸$_1$). If the preceding character is a space then remove the current character (▸$_2$). In only some cases we will be replacing *Char*'s value only to remove it entirely later, i.e. those cases where *Char*'s predecessor is a space. A general method says that if you can compute two actions sequentially and show the first is superseded by the second then you can get rid of the first.

To achieve the *ComputeSequentially* goal, we must distribute the call on REPLACE within the conditional:

```
...
  begin
   if predecessor(space, Char, Text)
    then begin
▸1          invoke REPLACE(Char newspace Text);
            invoke REMOVE(Char Text)
         end
    else invoke REPLACE(Char newspace Text);
  end
 ...
```

Finally, we can remove the first call to REPLACE ▸$_1$:

```
...
  begin
   if predecessor(space, Char Text)
    then invoke REMOVE(Char Text)
    else invoke REPLACE(Char newspace Text);
  end
  ...
```

E.3. Equivalence

Equivalence(C1|*construct*, C2|*construct*)

Achievement Condition: C1 is structurally equivalent to C2.

Goal Description: Equivalency here is based on structural or pattern-match semantics (see also the Lisp function **equals**): if C1 and C2 are two expressions in one-to-one correspondence, then C1 and C2 are equivalent. Note that in achieving this goal, there is no requirement that either C1 or C2 remain anchored; both may change into some new common form.

------------------------ **Examples of Use** ------------------------

Router References: 1.15, 2.10, 2.11, 4.5, 6.8, 6.12, 6.17, 6.18, 6.19

Example A

Router Reference: 4.5

Development context: when attempting to consolidate two structures, generally one or more of the components of each must be made equivalent. In consolidating the two demons at step 4.4, we find we must equivalence the two triggers (▶$_1$, ▶$_2$) of the two demons:

```
demon SET_SWITCH(switch)
▶1   trigger RANDOM()
     response ...

demon SET_SWITCH_WHEN_HAVE_CHANCE(switch, package)
▶2   trigger (package = first(PACKAGES_DUE_AT_SWITCH(*,switch))
                 and
              SWITCH_IS_EMPTY(switch))
     response ...
```

In this example, ▶$_2$ will be held constant (anchored) and ▶$_1$ changed to match it. This strategy

was chosen because of the general ease with which RANDOM can be specialized. After consolidation we have

```
demon SET_SWITCH(switch, package
    trigger (package = first(PACKAGES_DUE_AT_SWITCH(*,switch))
              and
            SWITCH_IS_EMPTY(switch))
     response ...
```

Example B

Router Reference: 2.10,2.11

Development context: equivalencing two compound structures is a frequently occurring goal. For instance, in step 2.10 we wish to make two demon argument lists equivalent: (*package.new*) is the first list and (*package*) the second. A useful method for achieving this goal employs a divide-and-conquer strategy by attempting to equivalence each subcomponent in a pairwise fashion. This leads to the equivalencing of *package.new* and *package* in step 2.11. Since each of these are primitive components, other methods will be employed (e.g. anchoring, renaming).

E.4. Factor

Factor(T|***template***, C|***construct***)

Achievement Condition: Factor all occurrences of T within C

Goal Description: As a development progresses, information tends to spread throughout the program. At certain points it is organizationally useful to regroup (factor) common structures.

The factor goal has two parameters: a template and a context. The template is a pattern with a special mechanism for marking formal parameters in the resulting definition. The context bounds the area in which the template will be matched[66].

---------------------------- **Examples of Use** ----------------------------

Router References: 6.5

Example A

Router Reference: 6.5

Following is a portion of the package router development, abstracted somewhat here for readability.

```
...
    if P
      then
       update packages_due of PACKAGES_DUE_AT_SWITCH(switch.current,$)
           to PACKAGES_DUE_AT_SWITCH(switch.current,*) minus package
      else
       loop Q do
        update packages_due of PACKAGES_DUE_AT_SWITCH(switch,$)
            to PACKAGES_DUE_AT_SWITCH(switch,*) minus package;
...
```

Using the template

[66] The *Isolate* goal can be viewed as a special case of the *Factor* goal where the context is exactly the expression to be factored.

```
    update packages_due of PACKAGES_DUE_AT_SWITCH(#switch[67], $)
        to PACKAGES_DUE_AT_SWITCH(#switch,*) minus #package
```

we can factor the two updates into a single new procedure:

```
...
      if P
       then invoke TRIM_PACKAGES_DUE_AT_SWITCH(package,
                                               switch.current)
       else
          loop Q
          do invoke TRIM_PACKAGES_DUE_AT_SWITCH(package, switch)

procedure TRIM_PACKAGES_DUE_AT_SWITCH(package, switch)
          update packages_due of PACKAGES_DUE_AT_SWITCH(switch,$)
            to PACKAGES_DUE_AT_SWITCH(switch,*) minus package;
```

The usefulness of factoring here will become apparent later in the development when maintenance code must be introduced at each change to PACKAGES_DUE_AT_SWITCH, before occurring in two locations, but now only one.

Example B

Router Reference: Heap Sort

The following is a portion of an intermediate state in the development of a heap sort algorithm suggested by Tim Standish:

```
...
procedure SiftUp(i,n)
 declare j: integer;
 begin
   if 2*i>n then Exit else j := 2*i;
   if 2*i<n then if C(2*i+1)>C(j) then j := 2*i+1;
   if C(j)>C(i) then
    begin
     invoke Exchange(C(j) C(i));
     invoke SiftUp(j n)
    end;
```

Factoring 2*i gives us

[67] In a factor template, #type.name signifies a formal parameter. The # will be removed in the definition.

```
...
Procedure SiftUp(i,n)
 declare j: integer;
 relation double_i(V|integer)
    definition V = 2*i;
 begin
    if double_i(*)>n then Exit else j := double_i(*);
   if double_i(*)<n then if C(double_i(*)+1)>C(j) then j:=double_i(*)+1;
   if C(j)>C(i) then
    begin
     invoke Exchange(C(j) C(i));
     invoke SiftUp(j n)
    end;
```

Further development yields

```
...
procedure SiftUp(i,n)
 declare j: integer;
 begin
   j := 2*i;
   if j>n then Exit;
   if j<n then if C(j+1)>C(j) then j :=j+1;
   if C(j)>C(i) then
    begin
     invoke Exchange(C(j) C(i));
     invoke SiftUp(j n)
    end;
```

E.5. Flatten

Flatten(C|*construct*)

Achievement Condition: No procedure calls or derived relation references exist in C.

Goal Description: The *Flatten* goal can be used for several different purposes:

- To explicate dependencies. For example, before maintaining a derived relation R, we must determine the set of base relations that R depends on (is defined in terms of). A simple way to determine the base set is to make all base relations explicit within R's body, i.e. *Flatten* any derived relations within R's body.

- To optimize. In general, optimizations cannot be carried out across definitional boundaries. If C is shown to be crucial to the performance of the program as a whole, then we may want to *Flatten* the procedure calling structure within C to allow local optimization to be carried out.

The methods used to flatten a context rely on either maintaining or unfolding defined objects. Hence, *Flatten* could be described as one or more postings of *Unfold* and/or *MaintainIncrementally*, making *Flatten* a vocabulary enriching, but unnecessary goal.

------------------------ **Examples of Use** ------------------------

Router references: 1.8, 5.3, 5.7

Example A

Router Reference: 1.8

Development context: the goal of step 1.7 is the incremental maintenance of the derived relation PREVIOUS_PACKAGE.

```
relation PREVIOUS_PACKAGE(prev_package | package)
   definition prev_package =
    (a package.previous ||
     package.previous immediately < last(PACKAGES_EVER_AT_SOURCE(*))
▸1            wrt PACKAGES_EVER_AT_SOURCE(*));
```

To maintain PREVIOUS_PACKAGE, we must determine when it changes, i.e. what relations it depends on. In this case, there is one: PACKAGES_EVER_AT_SOURCE ($\blacktriangleright_1$). However, PACKAGES_EVER_AT_SOURCE is a derived relation itself which may be defined in terms of still further relations. To explicate PREVIOUS_PACKAGES's base relations, a *Flatten* goal is posted at step 1.8. Note that if PACKAGES_EVER_AT_SOURCE was defined in terms of still further derived relations, these in turn would have to be flattened (see step 5.3).

E.6. Globalize

Globalize(C|***construct***)

Achievement Condition: C is to be moved out of the local context: local connections have been snipped; C is not part of an atomic.

Goal Description: Much work in a development involves moving structures from one place to another. In pulling some piece of code out of a particular context, we must make sure of several things:

- Any references to locally scoped variables within C should, *if possible*, be removed. If one or more variables resist removal, then C must be encapsulated and an argument defined for each local variable remaining.
- C cannot be part of an atomic. The statements of an atomic are treated as an indistinguishable action and cannot be spread out individually.

------------------------ Examples of Use ------------------------

Router Reference: 1.4, 5.12, 5.16

Example A

Router Reference: 1.4

Development context: at step 1.3, a goal is posted to *Isolate* a derived object ($\blacktriangleright_1$) found in the demon RELEASE_PACKAGE_INTO_NETWORK. The derived object makes reference to the variable *package.now*, locally scoped by the demon.

```
demon RELEASE_PACKAGE_INTO_NETWORK(package.new)
  trigger package.new:LOCATED_AT = the source
  response
    begin
      if
▸1          (the package.previous ||
              package.previous immediately before package.new
                  wrt PACKAGES_EVER_AT_SOURCE(*)
           ):DESTINATION ≠ package.new:DESTINATION
       then WAIT[];
      update :LOCATED_AT of package.new to (the source):SOURCE_OUTLET
    end;
```

If the reference to *package.new* is not eliminated, the resulting derived relation must include it as an argument.

Example B

Router Reference: 5.12

Development context: in this example we are trying to move a piece of code ▸2 out of a demon which is part of the environment (see *Purify*, section E.10).

```
demon CREATE_PACKAGE()
    trigger RANDOM()
    response
       atomic
         create package.new ||
             package.new:DESTINATION = a bin and
             package.new:LOCATED_AT = the source;
▸2         loop (switch ||
            MEMO_LOCATION_BIN(switch, package.new:DESTINATION))
            do update packages_due of PACKAGES_DUE_AT_SWITCH(switch,$)
              to PACKAGES_DUE_AT_SWITCH(switch,*) concat <package.new>
       end atomic;
```

Although the loop makes no reference to locally scoped variables, it is part of an atomic which prohibits it from being moved. To *Globalize* the loop, it must be removed from the atomic.

E.7. Isolate

Isolate(E|*expression*)

Achievement Condition: Replacement of E with reference to defined relation.

Goal Description: This goal reformulates some local embedded expression into a global one. This is generally the first step in moving the expression to a location where it can be further optimized. Note that the *Isolate* goal is a special case of *Factor* where the template must be a value returning expression and the context is the expression itself. In this sense, it is equivalent to a *Fold* in apllicative inaguage development systems (e.g. [Darlington 81]). We believe it occurs frequently enough as a speical case of factoring to be broken out separately.

------------------------ **Examples of Use** ------------------------

Router References: 1.3, 1.17, 3.3

Example A

Router Reference: 3.3

Development context: in section 3, we are concerned with the removal of the relation LAST_PACKAGE: only the destination of the last package is needed. The general strategy used is to remove all references to the relation, thus making the definition removable. There is only one reference to the relation:

```
...
        if LAST_PACKAGE(*):DESTINATION ≠ package.new:DESTINATION
            then invoke WAIT();
...
```

By posting an *Isolate* goal on the retrieval of the last package's destination, we can make this expression global.

```
...
        if LAST_PACKAGE_DESTINATION(*) ≠ package.new:DESTINATION
            then invoke WAIT();
...
relation LAST_PACKAGE_DESTINATION(last_destination| bin)
    definition last_destination = LAST_PACKAGE(*):DESTINATION;
...
```

The global computation, in the form of a derived relation, can now be moved to a location where further optimizations can be performed (see step 3.4).

Example B

Router Reference: `Line Draw`

Development context: Sproull presents the development of a line drawing algorithm which attempts to minimize the reliance on costly arithmetic operations such as multiplication and division. We will view the use of such operators as *specification freedoms* that must be mapped[68]. We are given the following portion of program for drawing a "straight line" between two points (0,0 and dx,dy) on a graphics screen[69]:

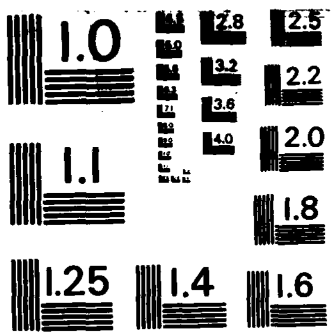
```
...
  loop x from 0 to dx
   do begin
      y := truncate([dy/dx] * x + 1/2);
      DISPLAY(x y)
   end;
...
```

Our goal is to map the *multiplication operation into an acceptable operation (e.g. addition) on* the final implementation hardware. The method we wish to use replaces the multiplication of the loop variable by a constant with a new expression only using addition (as residue, it leaves another expression involving multiplication that can be mapped later). The method expects that the multiplication has been isolated, i.e. it cannot work on embedded expressions.

[68]Note that Sproull's development is the algorithmic optimization type that we have disassociated from. However, the freedom mapping view makes it an illustrative example.

[69]The pseudo Pascal notation is Sproull's. The Gist version would replace variables with relations and assignments with inserts and updates.

AD-A139 918 AUTOMATING THE TRANSFORMATIONAL DEVELOPMENT OF SOFTWARE 3/3
VOLUME 2 APPENDICES(U) UNIVERSITY OF SOUTHERN
CALIFORNIA MARINA DEL REY INFORMATION S.. S F FICKAS
UNCLASSIFIED MAR 83 ISI/RR-83-109 NSF-MCS79-18792 F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Transformation RemoveMultiplication:

```
loop i from c1 to c2
 do begin
    z := c3 * i
    ...
 end;

=>

z := (c1 - 1) * c3;
loop i from c1 to c2
 do begin
    z := z + c3;
    ...
 end;
```

Using isolation leads us to the following state in which the RemoveMultiplication transformation can be applied:

```
...
 loop x from 0 to dx
  do begin
     t := [dy/dx] * x;
     y := truncate(t + 1/2);
     DISPLAY(x y)
  end;
...
```

Further in the same development, we reach the following state:

```
...
 t := 0;
 loop x from 0 to dx
  do begin
     s := t + 1/2;
     y := truncate(s);
     DISPLAY(x y)
     t := t + [dy/dx]
  end;
...
```

The goal is now the removal of the variable *t*. Again using isolation, in this case the reference to *t* in the computation of *s*, we get

```
relation s|REAL = t + 1/2;
...
  t := 0;
  loop x from 0 to dx
   do begin
      y := truncate(s);
      DISPLAY(x y)
      t := t + [dy/dx]
   end;
...
```

Finally, after computing s at each place it changes (see the goal MaintainIncrementally) we get

```
relation s | real;
...
  atomic
    t := 0;
    s := 0 + 1/2
  end atomic
  loop x from 0 to dx
   do begin
      y := truncate(s);
      DISPLAY(x y)
      atomic
        t := t + [dy/dx];
        s = s + [dy/dx]
      end atomic
   end;
...
```

which can be simplified into

```
relation s | real;
...
  s := 0 + 1/2
  loop x from 0 to dx
   do begin
      y := truncate(s);
      DISPLAY(x y)
      s = s + [dy/dx]
   end;
...
```

E.8. Map

Map(C|*construct*)

Achievement Condition: The *freedom* embodied by C has been mapped away.

Goal Description: A large part of the development of an abstract specification involves finding ways to remove specification freedoms which are not supported in the implementation language. What is considered a freedom is naturally dependent on the specification language being used and the final implementation language. The following are Gist specification freedoms: derived-relations, temporal reference, demonic computation, constraints and non-deterministic selection (see section 5.2.1 for further discussion). Depending on the implementation language, other freedoms might include recursi{n, parallelism, the associative relational data base and even multiplication (see example B in section E.7).

------------------------ **Examples of Use** ------------------------

Router References: 1.10, 4.1, 4.3, 4.7, 4.9, 4.10, 4.12, 4.13, 4.15, 4.16, 4.18, 5.1, 5.4, 5.5, 5.8, 6.1, 6.3, 6.6, 6.13, 6.15, 6.20

Example A

Router Reference: 5.4

Development context: LOCATION_ON_ROUTE_TO_BIN is one of the derived relations found in the specification:

```
relation LOCATION_ON_ROUTE_TO_BIN(LOCATION,BIN)
   definition
      case LOCATION of
       BIN    => LOCATION = BIN;
       PIPE   =>  LOCATION_ON_ROUTE_TO_BIN(
                    LOCATION:connection_to_switch_or_bin,BIN);
       SWITCH  => LOCATION_ON_ROUTE_TO_BIN(LOCATION:switch_outlet,BIN);
       SOURCE   => LOCATION_ON_ROUTE_TO_BIN(LOCATION:source_outlet,BIN);
      end case;
```

It is mapped away by remembering the router connections explicitly:

```
...
relation MEMO_LOCATION_BIN(location, bin);

demon INITIALIZE_MEMO_LOCATION_BIN()
   trigger: (start initialization_state)
   response
    begin
      loop B | BIN do insert MEMO_LOCATION_BIN(B, B);
      loop L | LOCATION ||
                        MEMO_LOCATION_BIN(L, B) and
                        L = L2:CONNECTION_TO_SWITCH_OR_BIN
             do insert MEMO_LOCATION_BIN(L2, B);
    end
...
```

Example B

Router Reference: 4.1

Development context: the constraint DID_NOT_SET_SWITCH_WHEN_HAD_CHANCE is a freedom which must be mapped:

```
constraint DID_NOT_SET_SWITCH_WHEN_HAD_CHANCE
  always prohibit ∃ package,switch ||
   (package:LOCATED_AT = switch
     and
    SWITCH_SET_WRONG_FOR_PACKAGE(switch,package)
     and
    ((package = first(PACKAGES_DUE_AT_SWITCH(*,switch))
       and
     SWITCH_IS_EMPTY(switch)) asof everbefore));
```

The method employed maps the constraint into a demon which triggers on one of the conjunctive arms of the constraint, and requires that the other two arms not hold. The trick here is choosing which arm to trigger on, i.e. whcich event allows the others to be avoided. The choice is currently left ot the user. The new demon is

```
...
demon SET_SWITCH_WHEN_HAVE_CHANCE(switch, package)
  trigger (package = first(PACKAGES_DUE_AT_SWITCH(*,switch))
                and
              SWITCH_IS_EMPTY(switch))
  response
    require (~(package:LOCATED_AT = switch
                     and
              SWITCH_SET_WRONG_FOR_PACKAGE(switch,package))
                 from ThisEvent[70]
                 until ~((package =
                             first(PACKAGES_DUE_AT_SWITCH(*,switch))
                           and
                         SWITCH_IS_EMPTY(switch)) asof everbefore))
```

We now must map this demon. The general strategy will be to consolidate this demon with the SET_SWITCH demon which controls the setting of switches. Note that the use of demons as intermediate mapping forms appears useful and is replected in the selection rule DemonsAreGood.

Example C

Router Reference: 4.18

Development context: at step 4.18, the update of a switch's setting is still in non-deterministic form:

```
update :SWITCH_SETTING of switch to switch:SWITCH_OUTLET
     where SWITCH_IS_EMPTY(switch)
                 and
           ~SWITCH_SET_WRONG_FOR_PACKAGE(switch,package);
```

The method employed will be to choose, deterministically, a setting that does not violate the attached constraints:

[70] i.e. the triggering of this demon.

```
...
    update :SWITCH_SETTING of switch to
           (pipe || pipe = switch:SWITCH_OUTLET
                    and
              LOCATION_ON_ROUTE_TO_BIN(pipe,
                                  package:DESTINATION));
```

E.9. MaintainIncrementally

MaintainIncrementally(R|*defined-relation*)

Achievement Condition: R recomputed *eagerly* (as opposed to lazy evaluation) in terms of the changes to the value upon which it is defined.

Goal Description: A derived relation R is defined in terms of another expression E. We can remove the need for E by making sure that R is maintained throughout the program. That is, wherever the value of E changes, we introduce code to incrementally update R.

------------------------- **Examples of Use** -------------------------

Router References: 1.8, 1.11, 1.18, 3.4, 5.2

Example A

Router Reference: 1.11

Development context: The goal of step 1.10 is to map the derived-relation PACKAGES_EVER_AT_SOURCE (or PEAS). There are several general strategies we wcan try: maintain the relation incrementalyy; unfold the relation where ever it is used (lazy evaluation). The relation PEAS is ideally suited for an incremental maintenance approach: packages are added to the end of the sequence one at a time.

```
...
relation PACKAGES_EVER_AT_SOURCE(package_seq|sequence of package)
   definition package_seq =
     ({package || (package:LOCATED_AT = the source) asof everbefore}
        ordered temporally by start (package:LOCATED_AT = the source));
...
```

The MaintainIncrementally goal posted at 1.11 triggers several competing methods. That is, the concept or general strategy of incremental maintenance was generalized into a goal with a set of methods or tactics for actually carrying it out. The method we will use introduces a demon which "watches" for relevant changes (a package becoming located at the source station) and does the necessary update to PEAS.

```
...
demon NOTICE_NEW_PACKAGE_AT_SOURCE(package.new)
  trigger package.new:LOCATED_AT = the source
  response
    update package_seq in PACKAGES_EVER_AT_SOURCE($)
        to PACKAGES_EVER_AT_SOURCE concat <package.new>;

relation PACKAGES_EVER_AT_SOURCE(package_seq|sequence of package);
...
```

Example B

Router Reference: 1.8

In step 1.8 we wish to incrementally maintain the relation PREVIOUS_PACKAGE:

```
...
relation PREVIOUS_PACKAGE(prev_package | package)
   definition prev_package =
     (a package.previous ||
        package.previous immediately < last(PACKAGES_EVER_AT_SOURCE(*))
         wrt PACKAGES_EVER_AT_SOURCE(*));
...
```

Instead of using a demon as in example A, we will employ a method which scatters maintenance code (▸$_2$) at every location within the program where the relation may change, i.e. where its base relation PACKAGES_EVER_AT_SOURCE changes. There is only one such location (▸$_1$) and that is found within NOTICE_NEW_PACKAGE_AT_SOURCE.

```
...
relation PREVIOUS_PACKAGE(prev_package | package);

demon NOTICE_NEW_PACKAGE_AT_SOURCE(package.new)
  trigger package.new:LOCATED_AT = the source
  response
    atomic
▶1      update package_seq in PACKAGES_EVER_AT_SOURCE($)
          to PACKAGES_EVER_AT_SOURCE concat <package.new>;
▶2      update prev_package in PREVIOUS_PACKAGE($)
          to (the package.previous ||
              package.previous immediately before
                last(PACKAGES_EVER_AT_SOURCE(*) concat <package.new>)
                 wrt PACKAGES_EVER_AT_SOURCE(*) concat <package.new>)
    end atomic
...
```

E.10. Purify

Purify(A|*action*)

Achievement Condition: A does not appear inside an uncontrollable portion of the spec.

Goal Description: During a development, the unfolding and maintaining of defined structures may lead to the introduction of code into portions of the specification which are uncontrolable. For instance, a specification may contain a model of the environmentin which the application program is to run. Code introduced intosuch uncontrollable portions must be moved to parts of the spec that are under control of the application program. We *Purify* a newly introduced action A by either 1) doing nothing if A is in the implementable portion of the spec (the goal is trivially satisfied) or 2) removing A from the uncontrollable portion.

························ **Examples of Use** ························

Router reference: 5.10, 5.14

Example A

Router Reference: 5.10

Development context: in the process of maintaining PACKAGES_DUE_AT_SWITCH in section 5 maintenance code ($\blacktriangleright_1$) is introduced into the demon CREATE_PACKAGE:

```
demon CREATE_PACKAGE()
    trigger RANDOM()
    response
      atomic
        create package.new ||
            package.new:DESTINATION = a bin and
            package.new:LOCATED_AT = the source;
▶1        loop (switch ||
          MEMO_LOCATION_BIN(switch, package.new:DESTINATION))
          do update packages_due of PACKAGES_DUE_AT_SWITCH(switch,$)
            to PACKAGES_DUE_AT_SWITCH(switch,*) concat <package.new>
      end atomic;
```

In step 5.10, we post a goal to *Purify* the new code. Since CREATE_PACKAGE is outside the implementable portion of the spec -- it is a part of the model of the environment -- the achievement of the goal rests on moving the code to an implementable part of the spec, in this case the demon RELEASE_PACKAGE_INTO_NETWORK.

E.11. Reformulate

Reformulate(C|***construct***, P|***pattern***)

Achievement Condition: A state is reached where C matches P

Goal Description: Using the *Reformulation* goal, the user can describe a goal state as a syntactic pattern. Such a general goal has great expressive power. In fact, we can express several other defined goals through the Reformulate goal: *Remove* given the empty state as a pattern; sometimes *Map* where the mapped state can be described by a syntactic pattern (e.g. derived-relations).

Over reliance on syntactic goal descriptions loses the development abstraction we strive for, i.e. an explicit vocabulary of goals for which specific methods can be developed. Currently, use of the Reformulate goal in a development is viewed as ad hoc: the pattern has not occurred enough to generalize into a new goal descriptor. As more experience is gained in developing programs using Glitter, we expect further pattern generalization to occur.

---------------------------- **Examples of Use** ---------------------------

Router References: 1.5, 1.13, 1.14, 1.16, 1.20, 2.12, 4.6, 6.9, 6.11

Example A

Router Reference: 1.5

Development context: Before a derived object is folded into a derived relation (i.e. *Isolated*), an attempt is made to remove as much linkage to the local context as possible (i.e. *Globalize*). In step 1.5, the local variable *package.new* is to be reformulated into a ***global-expression***, one which consists solely of relations and global objects. At step 1.6, this goal has been further reduced to reformulating the variable into an expression on PACKAGES←EVER←AT←SOURCE, namely last(PACKAGES_EVER_AT_SOURCE(*)). Having gotten this far, the system does not have the necessary theorem proving capability to show that these two expressions are equivalnet, and hence relies on the user to fill-in the last step.

Example B

Router Reference: `1.13, 1.14`

Development context: The goal of step 1.12 is to remove the reference to PACKAGES_EVER_AT_SOURCE from the following context:

```
▶1   (the package.previous ||
          package.previous immediately before
            last(PACKAGES_EVER_AT_SOURCE(*) concat <package.new>)
             wrt PACKAGES_EVER_AT_SOURCE(*) concat <package.new>)
```

The method chosen attempts to reformulate the derived object ▶$_1$ as a positional-retrieval on PACKAGES_EVER_AT_SOURCE which may prove easier to work with:

`goal-pattern: last(S|sequence)`

A method exists for reformulating derived objects of a certain type, namely ones that do a trivial binding:

`goal pattern: (x || x = last(S|sequence))`

Finally, a method exists for reformulating relative retrievals from a sequence into positional ones:

`goal pattern: x immediately before y wrt (S|sequence concat z)`

This last pattern can be matched directly against the current state.

Example C

Router Reference: `4.6, 6.9`

Development context: A general means of making two expressions equivalent is to hold one steady and reformulate the other. This crops up several places within the router development when two demon triggers need to be made equivalent. In the first, RANDOM must be reformulated as

package = first(PACKAGES_DUE_AT_SWITCH(*, *switch*)
and
SWITCH_IS_EMPTY(*switch*)

Here, a method which replaces a random event with a more specific event is chosen.

In the second, we must reformulate the relation reference SWITCH_IS_EMPTY(*switch*) as

package:LOCATED_AT = *switch*

Here, a method which unfolds the relation at its reference point is chosen.

E.12. Remove

Remove(S|*construct*, C|*construct*))

Achievement Condition: Structure S is removed from context C

Goal Description: The removal of structure S from context C may be motivated by any of the following:

1. S is deadwood; no use is made of S within C.
2. S is a component of some larger structure X; by stripping away all components of X, X can be removed (see 1 above).
3. C is a portion of the specification outside of which we have control.

.... **Examples of Use** ....

Router References: 1.1, 1.2, 1.12, 1.19, 1.21, 2.1, 2.2, 3.1, 3.2, 3.5, 5.11, 5.15

Example A

Router Reference: 1.1

Development context: section 1 of the router development centers on optimizing the relation (sequence) PACKAGES_EVER_AT_SOURCE. In particular, we only reference the last element of this sequence and hence, have no need for the entire history of packages ever entering the router. In step 1.1, the user states his desire to *Remove* this relation[71].

```
relation PACKAGES_EVER_AT_SOURCE(package_seq | sequence of package)
   definition package_seq =
      ({package || (package:LOCATED_AT = the source) asof everbefore}
         ordered temporally by start (package:LOCATED_AT = the source));
```

After a number of development steps, the above relation is removed from the spec, and as residue, the following two relations are left:

---

[71] Note the difference between mapping the relation and removing the relation. A mapping goal would be achieved when we had eliminated the derivation freedom from PACKAGES_EVER_AT_SOURCE (see step 1.9), the remove goal when the entire relation has been eliminated. In fact, the remove goal is a more specific case of the map goal: removing a derived relation entirely is one way of getting rid of the freedom.

```
relation PREVIOUS_PACKAGE(prev_package | package);
relation LAST_PACKAGE(last_package | package);
```

Example B

Router Reference: Text Preprocessor

Development context: in much the same way that the sequence **PACKAGES_EVER_AT_SOURCE** was unused in example A above, an action may be "unused". That is, there may be no references to its effects. In the text preprocessor development, we reach the following state (see example B, section E.2):

```
...
  begin
   if predecessor(space Char Text)
    then begin
▸1          invoke REPLACE(Char newspace Text);
            invoke REMOVE(Char Text)
        end
    else invoke REPLACE(Char newspace Text);
  end
 ...
```

The first replace procedure ▸$_1$ is wasted effort since the next action is to REMOVE the character. A goal is posted to *Remove* the call on REPLACE ▸$_1$.

Example C

Router Reference: 5.11

Development context: the above examples have dealt with removing a construct completely, i.e. from the entire spec. The *Remove* goal can also be used to remove a construct from a more specific context. For example, the effect of maintaining a derived relation is to place maintenance code *anywhere* in the spec where the relation might change. Some of these locations may be outside of the portion of the spec over which we have direct control, e.g. the portion of the spec that models the environment. Such is the case in the maintenance of PACKAGES_DUE_AT_SWITCH in section 5. Code is introduced into the demon CREATE_PACKAGE, part of the model of the router environment:

```
demon CREATE_PACKAGE()
    trigger RANDOM()
    response
      atomic
        create package.new ||
            package.new:DESTINATION = a bin and
            package.new:LOCATED_AT = the source;
▸1        loop (switch ||
            MEMO_LOCATION_BIN(switch, package.new:DESTINATION))
            do update packages_due of PACKAGES_DUE_AT_SWITCH(switch,$)
              to PACKAGES_DUE_AT_SWITCH(switch,*) concat <package.new>
      end atomic;
```

The maintenance code ▸$_1$ must be removed from CREATE_PACKAGE. While we could attempt to remove it from the entire spec, reasoning that this is one way of removing it here (this method is used in removing the same maintenance code from RELEASE_PACKAGE_INTO_NETWORK in section 5) the actual method chosen attempts to move the code out of CREATE_PACKAGE (and into the implementable portion), hence satisfying the goal.

E.13. Show

Show(P|*property*)

Achievement Condition: P asserted

Goal Description: The validity of many development methods rest on showing that certain properties hold in the current state of the program. Sometimes, one or more of the arguments to a property may be unbound. In these cases the task is to find some binding that makes the property hold. Below are listed the currently defined set of properties. Following each property is the locations in the router development where it is used as an applicability condition for a chosen method.

ACTION_IS_UNNOTICED(A|*action*) (1.22, 3.5)
An action A is unnoticed if either it has no effects or its effects are not used by any subsequent computation.

COMPUTATIONALLY_BETWEEN(E|*expression*, A1|*action*, A2|*action*) (2.5)
The expression E is computed after A1 is executed but before A2 is executed.

EVENT_BEFORE_EVENT(B|*event*, E|*event*) (4.14)
Event B occurs before event E.

FINITE_EXPLICATION(DR|*derived relation*) (5.4)
A finite number of explicit data base assertions will compute DR.

FUTURE_EVENT(F|*event*, C|*event*) (4.11)
Event F occurs after event C.

GENERALIZABLE_TRIGGER(T|*trigger*) (6.11)
The trigger (~∃ x || P(x)) can be replaced by ~P(x).
IMPLIED_BY(Q|*expression*, P|*expression*) (4.1, 4.9, 4.12)
Logical implication: P => Q.

INDIVIDUAL_START(D|*demon*) (6.2, 6.14)
If D has a conjunctive trigger, none of the arms ever occur simultaneously.

INTRODUCEABLE_VAR_NAME(V|*variable-name*, D|*declarative-construct*) (2.12, 6.19)
It is legal to introduce V as a variable declared in D, i.e. V does not conflict with any existing variables declared by D.

LAST_ACTION(A|*action*, E|*action-event*) (4.15)

E specifies the event of an action. Action A is the location of the last such event relative to current location.

MERGABLE_DEMONS(B1|*demon-body*, B2|*demon-body*, I|*ordering*) (2.9, 4.4, 6.7, 6.16)
The value of I is an interleaving of the two demon bodies B1,B2 suchthat valid behaviors remain.

NON_EMPTY_SPECIALIZATION(S|*expression*) (4.6)
E does not rule out all behaviors.

SEQUENTIAL_ORDERING(O|*ordering*, X|*atomic*) (2.7, 5.13, 5.16)
The statements of X have been ordered in O. The ordering is a valid sequentiation of the parallel atomic.

SUPERFLUOUS_ATOMIC(A|*atomic*) (2.7, 5.13, 5.16)
The statements in A do not need to be executed as a single step, i.e. no other construct (demon,constraint) gains or loses triggerings.

SWAPPABLE(A1|*action*, A2|*action*) (2.14)
A1 does not modify any data referenced by A2. A2 does not modify any data referenced by A1.

UNCHANGED_BETWEEN_EVENTS(P|*expression*, E1|*event*, E2|*event*) (2.5, 4.17)
The value of P does not change between the two events E1,E2.

UPDATE_VALUE_HOLDS(U|*update*, R|*relation-reference*) (2.4)
Given that U modifies the value of X to Y, this modification is unchanged (X's value is still Y) when R is computed.

VALUE_KNOWN(R|*relation-reference*, V|*object*) (2.3)
The value of R is V.

------------------------- **Examples of Use** -------------------------

In some cases, methods exist for asserting needed properties, and in some cases the necessary reasoning is beyond the reach of the system and the user is called to verify and assert the property. The examples below show both types of processes.

Example A

Router Reference: 1.22

Development context: at step 1.1, a goal is posted to remove the relation

PACKAGES←EVER←AT←SOURCE. The method chosen attempts to remove all reference to the relation. At step 1.21, a subgoal is posted to remove one such reference, an update of the relation.

```
update package_seq in PACKAGES_EVER_AT_SOURCE($)
  to PACKAGES_EVER_AT_SOURCE concat <package>)
```

The method chosen to remove the update relies on showing that the update is unnoticed, i.e. no other subsequent expression references the new value. At step 1.22, a *Show* goal is posted to show that the update is inedeed unnoticed. The method chosen to assert the necessary property is ShowDysteleological. This method takes a rather unsophisticated approach, asserting the property when no references exist to the updated relation, not just ones effected by the update.

Example B

Router Reference: 2.3

Development context: as in the previous example, at step 2.2 a reference to a particular relation, PREVIOUS_PACKAGE, is trying to be removed so that the relation itself can eventually be removed.

```
...
    if PREVIOUS_PACKAGE(*):DESTINATION ≠ package.new:DESTINATION
      then invoke WAIT[];
    ...
```

```
relation PREVIOUS_PACKAGE(prev_package | package);
```

The method chosen attempts to rpelace the reference with an actual value. To do this, the method posts a goal at step 2.3 to show that the value is known at the point of reference. The method chosen to assert the property relies on showing still another property: an update U of the relation to value V still holds at the reference. Showing, in general, that V is the relation's value at the reference is beyond the reasoning power of the system; the user is called on to assert the necessary property. Note that while the system was required to call on the user for assistance, the chosen method did a portion of the reasoning necessary to set a more specific context for the user.

E.14. Simplify

Simplify(C|*construct*)

Achievement Condition: No simplification transformation firings

Goal Description: The posting of this goal causes the transformations in the *simplification subcatalog* (see F.16) to be run until a quiescent state is reached, i.e. none of the transformations fire. C bounds the context in which simplification is to be carried out. Chapter 5 discusses simplification isuues in more detail.

------------------------ **Examples of Use** ------------------------

In the router development of appendix B, we have omitted the explicit posting of simplification steps in favor of textual comments.

Example A

Router Reference: `4.19, after unfold`

Development context: as happens in the development as a whole, simplification often requires a joint effort between user and machine. The simplification of many constructs relies on the user to provide sophisticated reasoning to prime the process. The simplification at step 4.19 is one such example. We are given the following state:

```
...
demon SET_SWITCH(switch, package)
  trigger package = first(PACKAGES_DUE_AT_SWITCH(*,switch))
                and
             SWITCH_IS_EMPTY(switch)
  response
     update :SWITCH_SETTING of switch to
              (pipe || pipe = switch:SWITCH_OUTLET
                         and
                 SWITCH_IS_EMPTY(switch)
                         and
▶1               ~(LOCATION_ON_ROUTE_TO_BIN(switch,
                                              package:DESTINATION)
                                and
                   ~LOCATION_ON_ROUTE_TO_BIN(pipe,
                                              package:DESTINATION));
```

The user can reason that *switch* is indeed on the route to *package's* destination (first term of ▶$_1$) and so can get rid of this term. However, the system currently has no indirect reasoning machinery, and hence cannot show that the definition of PACKAGES_DUE_AT_SWITCH requires that *switch be on the route to package's* destination. The user is required to get the process going:

STEP 4.20(*user*): *Manual*

```
MANUAL_REPLACE  LOCATION_ON_ROUTE_TO_BIN(switch, package:DESTINATION)
      with
        true
```

STEP 4.21(*user*): *Simplify* ▶$_1$

The resulting simplification process takes the following form:

Applying

(... ***true*** and ***term***) ⇒ (...***term***)

gives

...~(~LOCATION_ON_ROUTE_TO_BIN(*pipe*, *package*:DESTINATION));

Applying

~(***term***) ⇒ ~***term***

gives

```
   ...~~LOCATION_ON_ROUTE_TO_BIN(pipe, package:DESTINATION);
```

Applying

~~term ⇒ *term*

gives

```
...
demon SET_SWITCH(switch, package)
  trigger package = first(PACKAGES_DUE_AT_SWITCH(*,switch))
                    and
                  SWITCH_IS_EMPTY(switch)
  response
     update :SWITCH_SETTING of switch to
▸3                  (pipe || pipe =  switch:SWITCH_OUTLET
                                and
▸2                       SWITCH_IS_EMPTY(switch)
                                and
                         LOCATION_ON_ROUTE_TO_BIN(pipe,
                                              package:DESTINATION));
```

The same process can be carried out in removing the second conjuct arm ▸$_3$: replace it with **true** (again the user must provide the reasoning) and simplify the conjunction ▸$_2$. This gives us

---

```
...
demon SET_SWITCH(switch, package)
  trigger package = first(PACKAGES_DUE_AT_SWITCH(*,switch))
                    and
                  SWITCH_IS_EMPTY(switch)
  response
     update :SWITCH_SETTING of switch to
▸3                  (pipe || pipe =  switch:SWITCH_OUTLET
                                and
                         LOCATION_ON_ROUTE_TO_BIN(pipe,
                                              package:DESTINATION));
```

E.15. Swap

Swap(A1|***action***, A2|***action***)

Achievement Condition: A1 and A2, brothers in a begin/end block, are interchanged

Goal Description: allows the exchange of one or more actions within a begin/end block.

.... Examples of Use ....

Router references: 2.14

Example A

Router Reference: 2.14

Development context: our goal in step 2.13 is the computation of the update to LAST_PACKAGE (▸$_1$) after the reference to PREVIOUS_PACKAGE (▸$_2$).

```
demon RELEASE_PACKAGE_INTO_NETWORK(package.new)
  trigger package.new:LOCATED_AT = the source
  response
    begin
     update prev_package in PREVIOUS_PACKAGE($)
              to LAST_PACKAGE(*);
▸1    update last_package in LAST_PACKAGE($)
              to package.new
▸2    if PREVIOUS_PACKAGE(*):DESTINATION ≠ package.new:DESTINATION
       then WAIT[];
      update :LOCATED_AT of package.new to (the source):SOURCE_OUTLET
    end;
```

The method chosen attempts to *Swap* the two statements.

E.16. Unfold

Unfold(D|***definition***, R|***reference***)

Achievement Condition: D unfolded at reference point R

Goal Description: Given that our specification language gives us the ability to create global parameterized definitions (e.g. procedures, derived-relations, constraints, demons) and local implicit and explicit references to them, we would sometimes like to replace the local reference with the instantiated definition. The motivation for this step can be one of optimization (calls may be expensive), mapping (mapping a derived relation by unfolding it everywhere it is referenced, a demon everywhere it is triggered) or catalytic (the introduction of the definition in the local context allows further optimizations to occur). The Unfold goal requests that a particular global definition be instantiated at a particular reference point.

------------------------ **Examples of Use** ------------------------

Router References: 2.7, 5.6, 5.9, 5.13, 5.17, 6.4, 6.10, 6.21

Example A

Router Reference: 6.10

Development context: One means of reformulating a derived relation is to unfold it wherever referenced. Given the definition and use of SWITCH_IS_EMPTY below

```
relation SWITCH_IS_EMPTY(switch)
    definition ~∃ package || package:LOCATED_AT = switch;
...
    trigger SWITCH_IS_EMPTY(switch)
...
```

we can unfold SWITCH_IS_EMPTY to get

```
...
    trigger ~∃ package || package:LOCATED_AT = switch;
...
```

From this point, one more reformulation leads to the desired state.

Example B

Router Reference: 6.4

Development context: We can view the reference of a demon as a location that causes a state change which may cause the demon to trigger. Step 6.4 requests that the demon SET_SWITCH_WHEN_BUBBLE_PACKAGE be unfolded at such a location ▸$_1$:

```
demon SET_SWITCH_WHEN_BUBBLE_PACKAGE(switch)
  trigger ∃ package ||
            package = first(PACKAGES_DUE_AT_SWITCH(* switch))
  response...;

...

▸1      update packages_due of PACKAGES_DUE_AT_SWITCH(switch,$)
         to PACKAGES_DUE_AT_SWITCH(switch,*) concat <package.new>;
```

Appendix F
Method Catalog

F.1. Catalog Notation

The presentation of the Glitter development methods will be grouped around the individual Gold descriptors. Each method will be presented using the following format:

Method <name>
Goal: [<triggering goal>]^1
Filter: [<boolean expression>]^0
Action: [<development actions>]^1
[Short description of method.]
References: list of triggering steps for this method
End Method

A method's <name> is used to give it a unique textual handle and is intended to give a short description as well.

The references list points into the router development in appendix C. The items of this list are steps where the method was competing. Steps listed in boldface are ones where the method was chosen.

The rest of the fields conform to the description given in chapter 6.

F.2. Casify

| **Method** BinarySplit |

Goal: *Casify* C| ***+constraint***

Action: 1) *Apply* BINARY-SPLIT(C)

[+constraint P ⇒ +constraint Q implies P; +constraint ~Q implies P]

References: 4.8, 4.11, 4.14

| **End Method** |

| **Method** CasifyConjunctiveTrigger |

Goal: *Casify* D|***demon***

Filter: a) gist-type-of[T|trigger-of[D], ***conjunction***]

Action: 1) *Show* INDIVIDUAL_START(D)

2) *Apply* SPLIT_CONJUNCTIVE_TRIGGER(D, T)

[It may be easier to break a demon up into special cases and then trying to map. Make sure that no new triggerings are created.]

References: 6.2, 6.14

| **End Method** |

| **Method** CasifySuperTrigger |

Goal: *Casify* D|***demon***

Filter: a) trigger-of[T, D]

b) component-of[S|***supertype***, T]

Action: 1) *Apply* CASIFY_DEMON_SUPERTYPE(T, S)

[Spawn a separate demon for every subtype X of S.]

References: 5.18, 5.19

| **End Method** |

| **Method** PastInduction |

Goal: *Casify* C| **+constraint**

Action: 1) *Reformulate* C as **+constraint** P <u>during</u> E

2) *Show* EVENT_BEFORE_EVENT(B, E)

3) *Apply* PAST_INDUCTION_CASIFY(C, B)

[Use induction from some past state.]

References: 4.8, 4.11, 4.14

| **End Method** |

| **Method** CasifyFromUntilEverConstraint |

Goal: *Casify* C| **+constraint**

Action: 1) *Reforumlate* C as

P <u>from</u> E <u>until</u> <u>evermore</u>

2) *Apply* CASIFY_AS_NOW_AND_AFTER(C)

[You can show that C holds from E until everafter if you can show it holds at E and afte E.]

References: 4.8, 4.11, 4.14

| **End Method** |

| **Method** CasifyAroundEvent |

Goal: *Casify* C|**constraint**

Action: 1) *Reformulate* C as **constraint** P <u>after</u> E

2) *Show* FUTURE_EVENT(F, E)

3) *Apply* CASIFY_AROUND_EVENT(C, F)

[Choose some event F in the future and show that C holds before, during and after F.]

References: 4.8, 4.11, 4.14

| **End Method** |

| **Method** RefromulateAsMuxCase |

Goal: **Casify** X|*action*
Action: 1) Apply EMBED_IN_MUX_CASE(X)

{X ⇒ mux-case e c1:X c2:X ... cn:X}
References: TextPreprocessor
| **End Method** |

F.3. ComputeSequentially

| **Method** ConsolidateToMakeSequential |

Goal: *ComputeSequentially* A1|***action*** before A2|***action***
Filter: a) component-of[A1, D1|***demon***]
b) component-of[A2, D2|***demon***]
Action: 1) *Consolidate* D1 and D2

[It is easier to move actions around if they are in the same context.]
References: 2.8
| **End Method** |

| **Method** MoveOutOfAtomic |

Goal: *ComputeSequentially* B|***action*** before A|***action***
Filter: a) component-of[A, C|***atomic***]
Action: 1) *Unfold* C

[If you are trying to move A after B and A is in an atomic, unfold the atomic before attempting to continue.]
References: 2.6
| **End Method** |

| **Method** SwapUp |

Goal: *ComputeSequentially* Y before X

Filter: a) brother-of[X, Y]

Action: 1) *Swap* Y with predecessor of Y

[If you are trying to compute X after Y then move Y up.]

References: 2.13

| **End Method** |

F.4. Consolidate

| **Method** MergeDemons |

Goal: *Consolidate* D1|***demon*** and D2|***demon***

Action: 1) *Equivalence* trigger-of[D1] and trigger-of[D2]

2) *Equivalence* var-declaration-of[D1] and var-declaration-of[D2]

3) *Show* MERGABLE_DEMONS(D1, D2, I|***ordering***)

4) *Apply* DEMON_MERGE(D1, D2, I)

[You can consolidate two demons if you can show that they have the same local variables, the same triggering pattern and that they meet certain merging conditions.]

References: 2.9, 4.4, 6.7, 6.16

| **End Method** |

| **Method** ConsolidateEnumerationLoops |

Goal: **Consolidate** L1|***action*** **and** L2|***action***

Action: 1) **Reformulate** L1 **as** ***enumeration-loop***

2) **Reformulate** L2 **as** ***enumeration-loop***

3) **Equivalence** generator-of[*, L1] and generator-of[*, L2]

5) **Show** MERGABLE_LOOPS(L1, L2)

6) Apply MERGE_ENUMERATION_LOOPS(L1, L2)

{To consolidate two loops, make their generators equivalent and show that they are mergable.}

References: TextPreprocessor

| **End Method** |

| **Method** ConsolidateSimpleConds1 |

Goal: **Consolidate** C1|if P then A and C2|if Q then B

Action: 1) **Equivalence** P and Q

2) **Show** (hoare-axiom) P {A} Q

3) Apply MERGE_SIMPLE_CONDS_WITH_SAME_PREDICATE(C1, C2)

{If P then a;if P then b ⇒ If P then a;b under certain conditions.}

References: unused

| **End Method** |

| **Method** ConsolidateSimpleConds2 |

Goal: **Consolidate** C1|if P then A and C2|if Q then B

Action: 1) **Equivalence** A and B

2) **Show** (hoare-axiom) P {A} ~Q

3) Apply MERGE_SIMPLE_CONDS_WITH_SAME_ACTION(C1, C2)

{If P then a;if Q then a ⇒ If P or Q then a under certain conditions.}

References: TextPreprocessor

| **End Method** |

F.5. Equivalence

| **Method** EquivalenceCompoundStructures1 |

Goal: *Equivalence* S1|***compound-structure*** and
S2|***compound-structure***
Filter: a) gist-type-of[*, S1] = gist-type-of[*, S2]
b) fixed-structure[S1]
Action: 1) forall pairwise-component-of[C1,C2,S1,S2]
do Equivalence C1 and C2

{Divide-and-conquer: make the components of two fixed structures equivalent.}

References: unused

| **End Method** |

| **Method** EquivalenceCompoundStructures2 |

Goal: *Equivalence* S1|***compound-structure*** and
S2|***compound-structure***
Filter: a) gist-type-of[*, S1] = gist-type-of[*, S2]
b) ~fixed-structure[S1]
c) component-correspondence[S1, S2, C|***correspondence***]
Action: 1) forall correspondence-pairs[C, C1, C2]
do Equivalence C1 and C2

{Divide-and-conquer: make the components of two non-fixed structures equivalent.}

References: 2.10, 6.17

| **End Method** |

| **Method** Anchor1 |

Goal: *Equivalence* X *and* Y
Action: 1) *Reformulate* Y *as* X

[Try changing the second construct into something that matches the first.]

References: 1.15, 2.10, 2.11, 4.5, 6.8, 6.12, 6.18

| **End Method** |

| **Method** Anchor2

Goal: *Equivalence* X and Y

Action: 1) *Reformulate* X *as* Y

[Try changing the first construct into something that matches the second.]

References: 1.15, 2.10, 2.11, 4.5, 6.8, 6.12, 6.18

| **End Method**

| **Method** AddNewVar

Goal: *Equivalence* L1|***variable-list*** and L2|***variable-list***

Filter: a) length[L1] > length[L2]

b) member[V|***variable-declaration***, L1]

c) ~member[V, L2]

Action: 1) *Show* INTRODUCABLE-VAR-NAME(V, L2)

2) *Apply* INTRODUCE-NEW-VAR(V, L2)

[Try adding a new var to make the two lists equivalenct.]

References: 6.19

| **End Method**

F.6. Factor

| **Method** FactorDBMaintenanceIntoAction

Goal: *Factor* U|***db-maintenance*** in L

Action: 1) *Apply* CREATE_ACTION_FROM_TEMPLATE(U A)

2) forall match-pattern[U, W, L]

do *Apply* REPLACE_DBMAINTENACE_WITH_ACTION(W A)

[Create a new action A and then find all matches W in L and replace each with a call to the new action A.]

References: 6.5

| **End Method**

F.7. Flatten

| **Method** Flatten |

Goal: *Flatten* DR | ***derived-relation***
Action: 1) forall
reference-location[BR | ***derived-relation***, $, DR]
do *Map* BR

[Map all derived relations found in DR into simple ones.]
References: 1.9, 5.3, 5.7
| **End Method** |

F.8. Globalize

| **Method** GlobalizeAction |

Goal: *Globalize* A | ***action***
Filter: a) component-of[A, X | ***atomic***]
Action: 1) *Unfold* X

[You can't pull something out of an atomic: jitter.]
References: 5.12, 5.16
| **End Method** |

| **Method** GlobalizeDerivedObject |

Goal: *Globalize* DO | ***derived-object***
Action: 1) forall location-reference[V, $, DO]
suchthat V ≠ local-var-of[*, DO]
do Try *Reformulate* V as ***global-expression***

[Try changing all local variable references to global references.]
References: 1.4
| **End Method** |

F.9. Isolate

```
| Method  FoldGenericIntoRelation                                |

        Goal: Isolate X|expression
        Action:  1) Globalize X
                 2) Apply FOLD_INTO_RELATION(X)

      [Straightforward fold into derived-relation.]
   References: 1.3, 1.17, 3.3
| End Method                                                     |
```

F.10. MaintainIncrementally

```
| Method  ScatterMaintenanceForDerivedRelation                   |

        Goal: MaintainIncrementally DR|derived-relation
        Filter:  a) ~recursive[DR]
        Action:  1) Flatten body-of[DR]
                 2) forall location-reference[BR, S, DR]
              do forall location-reference[BR, L, spec)
                 do begin
                    Apply INTRODUCE_MAINTENANCE_CODE(DR L)
                    Purify L
                   end

      [To maintain a derived relation DR, find everywhere the base relations of DR are changed and
      stick code in to maintain. Make sure that all base relations are simple before maintenance and
      that all code is pure after.]
   References: 1.8, 1.11, 1.18, 3.4, 5.2
| End Method                                                     |
```

| **Method** IntroduceSeqMaintenanceDemon |

Goal: *MaintainIncrementally* DR | ***derived-relation***

Filter: a) gist-type-of[parameter-of[DR],
sequence]

Action: 1) *Reformulate* body-of[DR]
as **temporally-ordered-set-idiom**[72]
2) *Apply* INTRODUCE_SEQ_MAINTENANCE_DEMON(DR)

[One way of maintaining a derived sequence is to first change the definition into a temporal order -- ({x||P(x)asof everbefore} ordered temporally by P(x)) -- and then set up a demon with trigger P(x) to add elements.]

References: 1.11, 5.2

| **End Method** |

F.11. Map

| **Method** ShowNoChange |

Goal: *Map* C | *+constraint* ~(start of P)
between E1,E2

Action: 1) *Show* UNCHANGED_BETWEEN_EVENTS(P, E1, E2)
2) *Apply* REMOVE_UNCHANGED_CONSTRAINT(C)

[The direct approach.]

References: 4.16

| **End Method** |

---

[72] Patterns can be predefined and named. In this case, ({x||P(x) asof everbefore} ordered temporally by start P(x)).

| **Method** ChooseElementOfSet |

Goal: *Map* C|**+constraint**

Filter: a) gist-type-of[E|constraint-body[C], ***existential***]

Action: 1) *Show* ELEMENT_OF_SET(X, E)

2) *Apply* CHOOSE_ELEMENT(X, E)

[Try replacing the existential set with one of its elements.]

References: unused

| **End Method** |

| **Method** CasifyDemon |

Goal: *Map* D|***demon***

Action: 1) *Casify* D

2) forall case-of[X, D] do *Map* X

[Try mapping by case analysis.]

References: 4.3, 6.1, 6.3, 6.6, 6.13, 6.15, 6.19

| **End Method** |

| **Method** UnfoldDemon |

Goal: *Map* D|***demon***

Action: 1) forall trigger-location[D, L, spec]
do *Unfold* D at L

[To Map a demon, unfold it where appropriate.]

References: 4.3, 6.1, 6.3, 6.6, 6.13, 6.15, 6.20

| **End Method** |

| Method StoreExplicitly |

Goal: *Map* DR|***derived-relation***

Filter: a) STATIC(DR)

Action: 1) *Show* FINITE_EXPLICATION(DR)

2) *Apply* INITIALIZE_MEMO_RELATION(M, DR)

3) forall location-reference[DR, L, spec]

do *Apply* REPLACE-REF-WITH-MEMO(L, M)

4) *Apply* REMOVE_UNREFERENCED_RELATION(DR)

[You can explicitly compute a static derived relation given a finite number of resulting db insertions.]

References: 1.10, 5.1, 5.4, 5.5, 5.8

| End Method |

| Method UnfoldDerivedRelation |

Goal: *Map* DR|***derived-relation***

Action: 1) forall location-reference[DR, L, spec]

do *Unfold* DR at L

[One way of eliminating a derived relation is to unfold it at its reference points.]

References: 1.11 5.1, 5.4, 5.5, 5.8

| End Method |

| Method ComputeNewValue |

Goal: *Map* U|update X of Y to Z where P

Action: 1) *Apply*

COMPUTE_DERIVED_OBJECT_FROM_CONSTRAINT(U)

[Reformulate Z as derived object using P.]

References: 4.18

| End Method |

| **Method** MoveConstraintToAction |

Goal: *Map* C|*require*

Action: 1) *Reformulate* C as

require P at last E|*action-event*

2) *Show* LAST_ACTION(A|*action*, E)

3) *Apply* MOVE_CONSTRAINT_TO_ACTION(C, A)

[If a constraint C is on some action event E at A, attach the constraint to A.]

References: 4.7, 4.9, 4.10, 4.12, 4.13, 4.15, 4.16

| **End Method** |

| **Method** NotXUntilX |

Goal: *Map* R|+*constraint*

Action: 1) *Reformulate* R as +*constraint* P ... until E

2) *Show* IMPLIED_BY(P, ~E)

3) *Apply* REMOVE_VACUOUS_CONSTRAINT(R)

[P until E ⟹ true when ~E implies P]

References: 4.7, 4.9, 4.10, 4.12, 4.13, 4.15, 4.16

| **End Method** |

| **Method** TriggerImpliesConstraint |

Goal: *Map* R|*require*

Filter: a) component-of[R, D|*demon*]

Action: 1) *Reformulate* R as require P at *ThisEvent*

2) *Show* IMPLIED_BY(P, trigger-of[D])

3) *Apply* REMOVE_IMPLIED_REQUIREMENT(R)

[If a requirement is part of a demon, try showing that it is implied by the demon's trigger.]

References: 4.7, 4.9, 4.10, 4.12, 4.13, 4.15, 4.16

| **End Method** |

| Method CasifyPosConstraint |

Goal: *Map* C|+*constraint*

Action: 1) *Casify* C

2) forall case-of[X, C] do *Map* X

[Try mapping by case analysis.]

References: 4.7, 4.9, 4.10, 4.12, 4.13, 4.15, 4.16

| End Method |

| Method UnfoldConstraint |

Goal: *Map* C|*constraint*

Action: 1) forall location-violation[V, C] do *Unfold* C at V

[Find all places constraint might be violated and unfold maintenance code.]

References: unused

| End Method |

| Method MapConstraintAsDemon |

Goal: *Map* C|*constraint*

Action: 1) *Reformulate* C as always prohibit P

2) *Show* IMPLIED_BY(Q, P)

3) *Apply* REFORMULATE_CONSTRAINT_AS_DEMON(C, Q, D_{new})

4) *Map* D_{new}

[To map a prohibitive constraint, first choose some predicate Q that is always true when the constraint is violated, and then introduce a demon whose trigger is Q and whose body is a requirement of ~P.]

References: 4.1

| End Method |

| **Method** MaintainDerivedRelation |

Goal: *Map* DR | ***derived-relation***
Filter: a) ~static[DR]
Action: 1) *MaintainIncrementally* DR

{One way of mapping a derived relation is to maintain it explicitly.}

References: 1.10, 5.1, 5.4, 5.5, 5.8

| **End Method** |

| **Method** MapRandomToForwardEnum |

Goal: **Map** G | ***random-element-generator***
Action: 1) **Show** **no_successor_reliance**(G)
2) Apply REFINE_SET_ENUM_TO_FORWARD_SEQ(G)

{You can map a random (or ND) generator to a forward generator under certain conditions.}

References: TextPreprocessor

| **End Method** |

| **Method** MapRandomToBackwardEnum |

Goal: **Map** G | ***random-element-generator***
Action: 1) **Show** **no_predecessor_reliance**(G)
2) Apply REFINE_SET_ENUM_TO_BACKWARD_SEQ(G)

{You can map a random (or ND) generator to a backward generator under certain conditions.}

References: unused

| **End Method** |

| Method MapByConsolidation |

Goal: *Map* D|*demon*
Filter: a) match-pattern[*demon*, D2, spec]
b) D ≠ D2
Action: 1) *Consolidate* D and D2

[To map D, find some other demon D2 and consolidate.]

References: 4.3, 6.1, 6.3, 6.6, 6.13, 6.15, 6.19

| End Method |

F.12. Purify

| Method PurifyDemon |

Goal: *Purify* A|*action* in D|*demon*
Action: 1) *Remove* L from D

[Remove unpure statement L from D.]

References: 5.10, 5.14

| End Method |

F.13. Reformulate

| **Method** ReformLocalAsFirst |

Goal: *Reformulate* V|*variable* **as** ***global-expression***

Filter: a) patten-match[relation name (seq|sequence of type) def:,
R, *spec*]

b) domain-type-of[type, V]

Action: 1) *Reformulate* V **as** first(name(*))

[If you can find a sequence containing the same type of objects as V then you may be able to change V into a specific reference to the sequence.]

References: 1.5

| **End Method** |

| **Method** ReformLocalAsLast |

Goal: *Reformulate* V|*variable* **as** ***global-expression***

Filter: a) patten-match[relation name (seq|sequence of type) def:,
R, *spec*]

b) domain-type-of[type, V]

Action: 1) *Reformulate* V **as** last(name(*))

[If you can find a sequence containing the same type of objects as V then you may be able to change V into a specific reference to the sequence.]

References: 1.5

| **End Method** |

| **Method** ReformulateEverMoreAsDuring |

Goal: *Reformulate* X **as** (~Y during E)

Filter: a) gist-type-of[X, predicate]

Action: 1) *Reformulate* X **as** (~Y asof evermore)

2) *Show* IMPLIED_BY(Y, E)

3) *Apply* REFORM-EVERMORE-AS-UNTIL(X, E)

[(~Y asof evermore) ⇒ (~Y during E) where Y implies E]

References: unused

| **End Method** |

| **Method** ReformulateUntilAsEvermore |

Goal: *Reformulate* U|until P as asof evermore

Action: 1) *Show* NULL_OCCURRENCE(until-event[S])

2) *Apply* UNTIL_NEVER_TO_EVERMORE(S)

[P until never ⇒ P asof evermore]

References: unused

| **End Method** |

| **Method** ReformulateAsCondByEmbedding |

Goal: **Reformulate** X as if True then X

Action: 1) Apply EMBED_IN_COND(X)

[X ⇒ If True then X)]

References: TextPreprocessor

| **End Method** |

| **Method** RenameVar |

Goal: *Reformulate* V1|***variable-declaration*** as V2|***variable-declaration***

Filter: a) scoped-in[V1 S]

Action: 1) *Show* INTRODUCEABLE_VAR_NAME(V2, S)

2) *Apply* RENAME_VAR(V1, V2, S)

[Replace all occurrences of V1 with V2 in S after showing that V2 does not conflict with scoped variables already defined within S.]

References: 2.12

| **End Method** |

| **Method** ReformulateActionCall |

Goal: **Reformulate** AC|***action-call*** as P

Action: 1) Apply UNFOLD_ACTION_CALL(AC)

2) **Reformulate** AC as P

{If trying to reformulate an action call, unfold the body and try and reformulate it.}

References: TextPreprocessor

| **End Method** |

| **Method** ReformulateDerivedObject |

Goal: *Reformulate* DO|***derived-object*** *as* P

Action: 1) *Reformulate* body-of[DO]

as local-var-of[*, DO]=P

2) *Apply* UNFOLD_DERIVED_OBJECT(DO)

[(x || x = P) ⟹ P]

References: 1.13

| **End Method** |

| **Method** ReformulateDerivedRelation |

Goal: *Reformulate* RR|***relation-reference*** as X

Filter: a) gist-type-of[name-of[R, RR],

derived-relation]

Action: 1) *Unfold* R at RR

[Try reformulating the body as X.]

References: 6.9

| **End Method** |

| **Method** ReformulateRelativeRetrievalAsLast |

Goal: *Reformulate* RS | ***relative-sequence-retrieval***
as "x | ***object***=<u>last</u>(Seq | SEQUENCE)"

Action: 1) *Reformulate* RS *as*
"x <u>immediately</u> <u>before</u> y <u>wrt</u> (Seq <u>concat</u> z)"
2) *Equivalence* y *and* z
3) *Apply* CHANGE_TO_RETRIEVAL_OF_LAST(RS)

[x <u>immediately</u> <u>before</u> y <u>wrt</u> (Seq <u>concat</u> y) ⇒ x = <u>last</u>(Seq)]

References: 1.14

| **End Method** |

| **Method** ReformulateRelativeRetrievalAsFirst |

Goal: *Reformulate* RS | ***relative-sequence-retrieval***
as "x | ***object***=<u>first</u>(Seq | SEQUENCE)"

Action: 1) *Reformulate* RS *as*
"x <u>immediately</u> <u>after</u> y <u>wrt</u> (z <u>concat</u> Seq)"
2) *Equivalence* y *and* z
3) *Apply* CHANGE_TO_RETRIEVAL_OF_FIRST(RS)

[x <u>immediately</u> <u>after</u> y <u>wrt</u> (y <u>concat</u> Seq) ⇒ x = <u>first</u>(Seq)]

References: 1.14

| **End Method** |

| **Method** ReformulateAsObject |

Goal: *Reformulate* SR | ***last-retrieval*** as O | ***object***

Action: 1) *Reformulate* parameter-of[*, SR] as (S <u>concat</u> O)
2) *Apply* SIMPLIFY_LAST(SR)

[<u>last</u>(S <u>concat</u> O) ⇒ O]

References: 1.16, 1.20

| **End Method** |

| Method SpecializeRandom |

Goal: *Reformulate* X|RANDOM as Y

Action: 1) *Show* NON_EMPTY_SPECIALIZATION(Y)

2) *Apply* REPLACE_RANDOM_WITH_SPECIALIZATION(X Y)

[You can always replace RANDOM with a more speicialized event if you can show the new eventdoes not remove all choices.]

References: 4.6

| End Method |

| Method ReformulateExistentialTrigger |

Goal: *Reformulate* T|<u>trigger</u> ~∃ o||R(o) as R(o')

Action: 1) *Show* TRIGGER_GENERALIZABLE(T)

2) *Apply* GENERALIZE_TRIGGER(T)

[You can reformulate an existential trigger into a universally quantified one under certain conditions.]

References: 6.11

| End Method |

F.14. Remove

| Method RemoveFromDemon |

Goal: *Remove* A|*action* from D|*demon*

Action: 1) *Globalize* A

2) forall trigger-location[D2|*demon*, body-of[*, D], *spec*]
do *Apply* MOVE_STATEMENT_TO_DEMON(A, D2)

[Find all demons that trigger from D and move the action A there.]

References: 5.11, 5.15

| End Method |

| **Method** RemoveRelation |

Goal: *Remove* R|***relation*** from ***spec***

Action: 1) forall reference-location[R,RR,***spec***]
do *Remove* RR from ***spec***
2) *Apply* REMOVE_UNREFERENCED_RELATION(R)

[You can remove a relation if you can remove all references to it.]

References: 1.1, 2.1, 3.1

| **End Method** |

| **Method** ReplaceRefWithValue |

Goal: *Remove* RR|***base-relation-reference***

Action: 1) *Show* VALUE_KNOWN(R, V)
2) *Apply* REPLACE_REF_WITH_VALUE(R V)

[One way of getting rid of a non-derived-relation reference is to replace it with its value.]

References: 1.12, 1.19, 2.2, 3.2

| **End Method** |

| **Method** MegaMove |

Goal: *Remove* RR|***relation-reference*** from ***spec***

Filter: a) component-of[RR, Y|***expression***]

Action: 1) *Isolate* Y in DR|***derived-relation***
2) *MaintainIncrementally* DR

[Remove the relation-reference RR by moving it directly after the locations it is assigned.]

References: 1.2, 1.12, 1.19, 2.2, 3.2

| **End Method** |

| Method PostionalMegaMove |

Goal: *Remove* RR|***relation-reference*** from ***spec***

Filter: a) component-of[RR, Y|***expression***]

b) gist-type-of[***sequence***, argument-of[*, RR]]

Action: 1) *Reformulate* Y as PR|***positional-retrieval***

2) *Isolate* PR in DR|***derived-relation***

3) *MaintainIncrementally* DR

[One way of getting rid of a reference to a sequence is to reformulate it as part of a positional retrieval, and then megamove it.]

References: 1.2, 1.12, 1.19, 2.2, 3.2

| End Method |

| Method RemoveVariable |

Goal: *Remove* V|***variable*** from S|***scope***

Action: 1) forall reference-location[V,VR,S]

do *Remove* VR from S

2) *Apply* REMOVE_UNREFERENCED_VARIABLE(V)

[You can remove a variable if you can remove all references to it.]

References: TextPreprocessor

| End Method |

| Method RemoveByObjectizingContext |

Goal: *Remove* RR|***relation-reference*** from spec

Filter: a) component-of[RR, Y|***expression***]

Action: 1) *Reformulate* Y as object

[One way of getting rid of a relation reference which is embedded in context Y is to reformulate Y as an explicit object.]

References: 1.2, 1.12, 1.19, 2.2, 3.2

| End Method |

| **Method** EmptyAndRemove |

Goal: **Remove** S

Filter: a) compound-structure S

Action: 1) forall immediate-component-of[X, S]
do **Remove** X
2) Apply REMOVE_EMPTY_STRUCTURE(*S*)

{Remove a compound strucutre S by removing each of its components X.}

References: unused

| **End Method** |

| **Method** RemoveUnusedAction |

Goal: **Remove** A | *action*

Action: 1) **Show** action_is_unnoticed(A)
2) Apply REMOVE-UNNOTICED-ACTION(A)

{Show that the current action is either not used or superseded by a subsequent action.}

References: 1.21, 3.5, 5.11, 5.15

| **End Method** |

| **Method** ReplaceVariableWithValue |

Goal: **Remove** VR | *variable-reference*

Action: 1) **Show**(value_is_known(VR V | *object*)
2) Apply REPLACE_VARIABLE_WITH_VALUE(VR V)

{If a variable's value is known fill it in.}

References: TextPreprocessor

| **End Method** |

| **Method** BabyWithBathWater |

Goal: **Remove** X

Filter: a) X component-of Y

Action: 1) **Remove** Y

{One drastic method of removing X is to remove strucutre X is embedded in.}

References: 1.2, 1.12, 1.19, 1.21, 2.2, 3.2, 3.5, 5.11, 5.15

| **End Method** |

F.15. Show

| **Method** ConjunctImpliesConjunctArm |

Goal: Show X|*conjunction* implies Y

Filter: a) unbound[Y]

b) conjuct-arm[A|*logical-expression*, X]

Action: 1) *Assert* X implies A

[(P_1 and P_2 and ...P_n) implies P_i]

References: 4.2

| **End Method** |

| **Method** ShowDysteleological |

Goal: Show action_is_unnoticed(U|*update*)

Filter: a) update-relation-of[R, U]

b) ~location-reference[R, $, spec]

Action: 1) *Assert* action_is_unnoticed(U)

[If you are trying to show that an update is unnoticed, show that it is never referenced.]

References: 1.22

| **End Method** |

| **Method** ShowUpdateGivesValue

Goal: *Show* VALUE_KNOWN(R | ***relation-reference***, V)

Filter: a) match-pattern[***update***, U, ***spec***]

b) name-of[R] = update-relation-of[*, U]

Action: 1) *Show* UPDATE_VALUE_HOLDS(U, R)

2) *Assert* VALUE_KNOWN(R, new-value-of[*, U])

[Find the last update of R and show that the newvalue is still valid.]

References: 2.3

| **End Method**

| **Method** ShowNewValueStillValid

Goal: *Show* UPDATE_VALUE_HOLDS(U | ***update***, R | ***relation reference***)

Filter: a) name-of[R] = update-relation-of[*, U]

Action: 1) *Show*

UNCHANGED_BETWEEN_EVENTS(new-value-of[*, U], U, R)

3) *Assert* UPDATE_VALUE_HOLDS(U, R)

[To show that the new update value is still around at R, show that the update value has not been changed before R.]

References: 2.4

| **End Method**

| **Method** MoveInterveningUpdate

Goal: *Show* UNCHANGED_BETWEEN_LOCATIONS(V | ***relation reference***,
U | ***update***,
R | ***relation reference***)

Filter: a) pattern-match[***update***, L, ***spec***]

b) update-relation-of[V, L]

Action: 1) *Show* COMPUTATIONALLY-BETWEEN[L, U, R]

2) *ComputeSequentially* R before L

[If an intervening update of V exists, move it after R.]

References: 2.5

| **End Method**

F.16. Simplify

In this section, we list the transformations that make up the simplification subcatalog. For further details, see section E.14.

Simplifying a conjunction

```
(and) ➡ true
(and ... false ...) ➡ false
(and p) ➡ p
(and ... true ...) ➡ (and ...)
(and ... p ... p ...) ➡ (and ... p ...)
(and ... (and p q r) ...) ➡ (and ... p q r ...)
(and ... p ... ~p ...) ➡ false,
```

Simplifying a disjunction

```
(or) ➡ True
(or ... true ...) ➡ true
(or p) ➡ p
(or ... false ...) ➡ (or ...)
(or ... p ... p ...) ➡ (or ... p ...)
(or ... (or p q r) ...) ➡ (or ... p q r ...)
(or ... p ... ~p ...) ➡ (or ... true ...)
```

Simplifying a negation

```
(not (not p)) ➡ p
(not true) ➡ false
(not false) ➡ true
```

Simplifying a conditional

(cond true → a ...) ⇒ a

(cond) ⇒ empty

(cond ... false → a ...) ⇒ (cond ...)

(cond ... true → a ...) ⇒ (cond ... true → a)

(cond p → (cond q → a)) ⇒ (cond p and q → a)

F.17. Swap

| **Method** SwapStatements |

Goal: *Swap* A with B

Action: 1) *Show* SWAPPABLE(A B)

2) *Apply* SWAP_STATEMENTS(A B)

[A:B ⇒ B:A under certain conditions.]

References: 2.14

| **End Method** |

F.18. Unfold

| **Method** ScatterComputationOfDerivedRelation |

Goal: *Unfold* DR|***derived-relation*** at L

Filter: a) location-reference[DR, L, $]

Action: 1) *Apply* UNFOLD_COMPUTATION_CODE(DR L)

2) *Purify* L

[To unfold a derived relation DR at a reference point, stick in code to compute it and make sure L is within implementable portion of spec.] .

References: 4.18, 5.6, 5.9, 6.10, 6.16

| **End Method** |

| **Method** ScatterComputationOfDemon |

Goal: *Unfold* D|***demon*** at L

Filter: a) trigger-location[D, L, S]

Action: 1) *Apply* UNFOLD_DEMON_CODE(D L)

2) *Purify* L

[To unfold a demon D at a trigger point, stick in code to compute it and make sure L is within implementable portion of spec.]

References: 6.4, 6.21

| **End Method** |

| **Method** UnfoldAtomic |

Goal: *Unfold* A|***atomic***

Action: 1) *Show* SEQUENTIAL-ORDERING(O|***ordering***, A)

2) *Show* SUPERFLUOUS_ATOMIC(A)

3) *Apply* UNFOLD-ATOMIC(A, O)

[You can unfold an atomic if you can show that there exists some valid sequential ordering of the statements and that no demonic or inferencing processes will be effected.]

References: 2.7, 5.13, 5.17

| **End Method** |

| **Method** UnfoldSimpleSB |

Goal: Unfold SB|begin S end

Action: 1) Apply UNFOLD_SIMPLE_NESTED_BLOCK(SB)

{...begin s end... ⇒ ...s...}

References: TextProeprocessor

| **End Method** |

Appendix G
Selection Catalog

G.1. Catalog Notation

Selection rules will be presented using the following format:

Selection Rule <name>
    IF: [<selection expression>][1]
    THEN: [<selection action>][1]
        [optional comments]
  References: list of steps where rule used in selection process
End Selection Rule

A rule's <name> is used to give it a unique textual handle and is intended to give a short description as well.

The references list points into the router development in appendix C. The items of the list are steps in which the rule played an active part in selecting a method.

For an explanation of the remaining fields, see chapter 7.

The selection rules are organized in the following manner:

- *Method Specific Rules*: grouped here as in appendix F, around the set of development goals. Each development method in appendix F will be listed here along with a list of steps where it was competing; bold faced steps mark steps in which the method was the one finally selected. Following each method are the selection rules pertaining to it (possibly none).
- *Action Ordering Rules*: listed after specific method.
- *Method Ordering Rules*: listed at the end of each goal section.

- *Problem Solving Resource Rules*: listed in section G.19.
- *General Rules*: listed in section G.20.

G.2. Casify

BinarySplit (4.8, 4.11, 4.14)

```
| SelectionRule   *BinarySplit1                          |
       IF  a) *BinarySplit is a candidate
           b) Good choice for Q is known
       THEN   +2
       [Good choice if have a Q in mind.]
| End Selection Rule                                     |
```

```
| SelectionRule   *BinarySplit2                          |
       IF  a) *BinarySplit is a candidate
           b) Good choice for Q is unknown
       THEN   -2
       [Bad choice if don't have a Q in mind.]
       References: 4.8, 4.11, 4.14
| End Selection Rule                                     |
```

CasifyConjunctiveTrigger (6.2, 6.13)

CasifySuperTrigger (5.18, 5.19)

PastInduction (4.8, 4.11, 4.14)

CasifyFromUntilEverConstraint (4.8, 4.11, 4.14)

CasifyAroundEvent (4.8, 4.11, 4.14)

RefromulateAsMuxCase (TextPreprocessor)

G.3. ComputeSequentially

ConsolidateToMakeSequential (2.8)

```
| SelectionRule  *ConsolidateToMakeSequential                |
      IF  a) ConsolidateToMakeSequential is a candidate
      THEN   +2
      References: 2.8
| End Selection Rule                                          |
```

MoveOutOfAtomic (2.6)

```
| SelectionRule  *MoveOutOfAtomic                             |
      IF  a) MoveOutOfAtomic is a candidate
      THEN   +2
      References: 2.6
| End Selection Rule                                          |
```

SwapUp (2.13)

```
| SelectionRule  *SwapUp                                      |
      IF  a) SwapUp is a candidate
      THEN   +2
      References: 2.13
| End Selection Rule                                          |
```

G.4. Consolidate

MergeDemons (2.9, 4.4, 6.7, 6.15)

```
| SelectionRule  *MergeDemons                                          |
      IF   a) MergeDemons is a candidate
      THEN   +5
      References: 2.9, 4.4, 6.7, 6.15
| End Selection Rule                                                   |
```

```
| SelectionRule  TriggersAlmostEquiv                                   |
      IF   a) MergeDemons is selected
           b) Triggers differ only in variable renaming
      THEN   action-2 > action-1
      [The first goal will fall-out as side-effect of second.]
| End Selection Rule                                                   |
```

ConsolidateEnumerationLoops (TextPreprocessor)

ConsolidateSimpleConds1 (unused)

ConsolidateSimpleConds2 (TextPreprocessor)

G.5. Equivalence

EquivalenceCompoundStructures1

```
| SelectionRule  *EquivalenceCompoundStructures1                       |
      IF   a) EquivalenceCompoundStructures1 is a candidate
      THEN   +5
| End Selection Rule                                                   |
```

EquivalenceCompoundStructures2 (2.10, 6.12, 6.17)

```
| SelectionRule  *EquivalenceCompoundStructures2                    |
      IF  a) EquivalenceCompoundStructures2 is a candidate
      THEN   +2
      References: 2.10, 6.12, 6.17
| End Selection Rule                                                |
```

Anchor1 (1.15, 2.10, 2.11, 4.5, 6.8, 6.12, 6.18)

```
| SelectionRule  *Anchor1a                                          |
      IF  a) Anchor1 is candidate
          b) X|object
      THEN   +2
      References: 2.4, 6.12, 6.18
| End Selection Rule                                                |
```

```
| SelectionRule  *Anchor1b                                          |
      IF  a) Anchor1 is candidate
          b) Y|RANDOM
      THEN   +5
| End Selection Rule                                                |
```

```
| SelectionRule  *Anchor1c                                          |
      IF  a) Anchor1 is candidate
          b) Y|derived-relation-reference
          c) Defintion of Y reformulatable as X
      THEN   +2
      References: 6.8
| End Selection Rule                                                |
```

]

Anchor2 (1.15, 2.10, 2.11, 4.5, 6.8, 6.12, 6.18)

```
| SelectionRule  *Anchor2a                                  |
        IF  a) Anchor2 is candidate
            b) Y|object
        THEN    +2
        References: 1.15, 2.11, 6.12, 6.18
| End Selection Rule                                        |
```

```
| SelectionRule  *Anchor2b                                  |
        IF  a) Anchor2 is candidate
            b) X|RANDOM
        THEN    +5
        References: 4.5
| End Selection Rule                                        |
```

```
| SelectionRule  *Anchor2c                                  |
        IF  a) Anchor2 is candidate
            b) X|derived-relation-reference
            c) Defintion of X reformulatable as Y
        THEN    +2
| End Selection Rule                                        |
```

AddNewVar

```
| SelectionRule  *AddNewVar                                 |
        IF  a) AddNewVar is candidate
        THEN    +2
| End Selection Rule                                        |
```

Method Ordering Rules

```
| SelectionRule  EquivVars1                                            |
      IF  a) Method *Anchor1 is a good candidate
          b) Method *Anchor2 is a good candidate
          c) X and Y are variable names
      THEN   Rely on user to choose
      [The manipulation of names is viewed as important and currently rests in the hands of
      the user.]
      References: 2.11, 6.12, 6.18
| End Selection Rule                                                   |
```

if correspondecne 1 has more type matches than corresp 2 then choose first

if corresp 1 has more usage matches (trigger vars) than corresp 2 then choose first.

if tried equivcompst before try addnewvar now else vice versa

G.6. Factor

FactorDBMaintenanceIntoAction (6.5)

```
| SelectionRule  *FactorDBMaintenanceIntoAction                        |
      IF  a) FactorDBMaintenanceIntoAction is a candidate
      THEN   +2
      References: 6.5
| End Selection Rule                                                   |
```

G.7. Flatten

Flatten (1.9, 5.3, 5.7)

```
| SelectionRule  *Flatten                                              |
      IF  a) Flatten is a candidate
      THEN   +2
      References: 1.9, 5.3, 5.7
| End Selection Rule                                                   |
```

G.8. Globalize

GlobalizeAction (5.10, 5.15)

```
| SelectionRule  *GlobalizeAction                      |
      IF  a) GlobalizeAction is a candidate
      THEN   +2
      References: 5.10, 5.15
| End Selection Rule                                   |
```

GlobalizeDerivedObject (1.4)

```
| SelectionRule  *GlobalizeDerivedObject               |
      IF  a) GlobalizeDerivedObject is a candidate
      THEN   +2
      References: 1.4
| End Selection Rule                                   |
```

G.9. Isolate

FoldGenericIntoRelation (1.3, 1.17, 3.3)

```
| SelectionRule  *FoldGenericIntoRelation              |
      IF  a) FoldGenericIntoRelation is a candidate
      THEN   +2
      [If applicable, use it.]
      References: 1.3, 1.17, 3.3
| End Selection Rule                                   |
```

G.10. MaintainIncrementally

ScatterMaintenanceForDerivedRelation (1.8, 1.11, 1.18, 3.4, 5.2)

```
| SelectionRule  *ScatterMaintenanceForDerivedRelation                |
        IF  a) ScatterMaintenanceForDerivedRelation is a candidate
        THEN   +2
        References: 1.8, 1.11, 1.18, 3.4, 5.2
| End Selection Rule                                                  |
```

IntroduceSeqMaintenanceDemon (1.11, 5.2)

```
| SelectionRule  *IntroduceSeqMaintenanceDemon                        |
        IF  a) IntroduceSeqMaintenanceDemon is a candidate
        THEN   +1
        References: 1.11, 5.2
| End Selection Rule                                                  |
```

Method Ordering Rules

```
| SelectionRule  MaintDR1                                             |
        IF  a) IntroduceSeqMaintenacneDemon is a good candidate
            c) ScatterMaintenanceForDerivedRelation is a good candidate
            d) DR has a complex definition
        THEN   ScatterMaintenanceForDerivedRelation
                                  > IntroduceSeqMaintenacneDemon
        [A complex definition means a large number of new demons must be introduced.]
        References: 5.2
| End Selection Rule                                                  |
```

G.11. Map

ShowNoChange (4.16)

```
| SelectionRule   *ShowNOChange                                        |
      IF  a) ShowNoChange is a candidate
      THEN    +2
      References: 4.16
| End Selection Rule                                                    |
```

ChooseElementOfSet (unused)

CasifyDemon (4.3, 6.1, 6.3, 6.6, 6.13, 6.15, 6.19)

```
| SelectionRule   *CasifyDemon                                         |
      IF  a) CasifyDemon is a candidate
          b) D has a conjunctive trigger
          c) One or more arms of the trigger are observable events
          d) One or more arms of the trigger are unobservable events
      THEN    +2
      [Different strategies for each so break out.]
      References: 6.1, 6.13
| End Selection Rule                                                    |
```

UnfoldDemon (4.3, 6.1, 6.3, 6.6, 6.13, 6.15, 6.19)

```
| SelectionRule   *UnfoldDemon                                         |
      IF  a) UnfoldDemon is a candidate
      THEN    +1
      [Try if nothing else looks good.]
      References: 4.3, 6.1, 6.3, 6.6, 6.13, 6.15, 6.19
| End Selection Rule                                                    |
```

StoreExplicitly (5.4)

| SelectionRule *StoreExplicitly |

IF a) StoreExplicitly is candidate

THEN +2

References: 5.4

| End Selection Rule |

MapByConsolidation (4.3, 6.1, 6.3, 6.6, 6.13, 6.15)

| SelectionRule *MapByConsolidation1 |

IF a) MapByConsolidation is a candidate

b) D does not trigger on an observable event

c) D2 triggers on an observable event

THEN +1

References: 4.3, 6.1, 6.3, 6.6, 6.13

| End Selection Rule |

| SelectionRule *MapByConsolidation2 |

IF a) MapByConsolidation is a candidate

b) D2 triggers randomly

THEN +2

References: 4.3, 6.1, 6.3, 6.6, 6.13, 6.15

| End Selection Rule |

| SelectionRule *MapByConsolidation4 |

IF a) MapByConsolidation is a candidate

b) D2 is not within implementable portion

THEN -2

References: 4.3, 6.1, 6.3, 6.6, 6.13, 6.15

| End Selection Rule |

```
| SelectionRule   *MapByConsolidation5                                 |
       IF   a) MapByConsolidation is a candidate
            b) D1 and D2 are case-brothers
       THEN   -2
       [Unlikely will want to re-join previously split cases.]
       References: 6.3
| End Selection Rule                                                   |
```

```
| SelectionRule   *MapByConsolidation6                                 |
       IF   a) MapByConsolidation is a candidate
            b) D1 and D2 triggers are "trivially" different
       THEN   +2
       [i.e. if only differ in variable naming]
       References: 6.15
| End Selection Rule                                                   |
```

UnfoldDerivedRelation (1.10, 5.1, 5.4, 5.5, 5.8)

```
| SelectionRule   *UnfoldDerivedRelation1                              |
       IF   a) UnfoldDerivedRelation is candidate
            b) DR is not recursive
       THEN   +2
       References: 1.10, 5.1, 5.5, 5.8
| End Selection Rule                                                   |
```

```
| SelectionRule   *UnfoldDerivedRelation2                              |
       IF   a) UnfoldDerivedRelation is candidate
            b) DR is recursive
       THEN   -2
       References: 5.4
| End Selection Rule                                                   |
```

ComputeNewValue (4.18)

MoveConstraintToAction (4.7, 4.9, 4.10, 4.12, 4.13, 4.15, 4.16)

NotXUntilX (4.7, 4.9, 4.10, 4.12, 4.13, 4.15, 4.16)

TriggerImpliesConstraint (4.7, 4.9, 4.10, 4.12, 4.13, 4.15, 4.16)

CasifyPosConstraint (4.7, 4.9, 4.10, 4.12, 4.13, 4.15, 4.16)

UnfoldConstraint (4.1)

```
| SelectionRule  *UnfoldConstraint                              |
       IF  a) UnfoldConstraint is a candidate
           b) Backtracking solution is possible
       THEN   +2
| End Selection Rule                                            |
```

MapConstraintAsDemon (4.1)

```
| SelectionRule  *MapConstraintAsDemon                          |
       IF  a) MapConstraintAsDemon is a candidate
           b) A predictive solution is possible
       THEN   +2
       References: 4.1
| End Selection Rule                                            |
```

MaintainDerivedRelation (1.10, 5.1, 5.5, 5.8)

```
| SelectionRule  *MaintainDerivedRelation                       |
       IF  a) MaintainDerivedRelation is candidate
       THEN   +2
       References: 1.10, 5.1, 5.5, 5.8
| End Selection Rule                                            |
```

MapRandomToForwardEnum (TextPreprocessor)

MapRandomToBackwardEnum (unused)

Method Ordering Rules

```
| SelectionRule  MapDR1a                                              |
      IF  a) StoreExplicitly is a good candidate
          b) Number of refs * recompute cost is more costly than
          number of explicit insertions
      THEN   StoreExplicitly > UnfoldDerivedRelation
      References: 5.4
| End Selection Rule                                                  |
```

```
| SelectionRule  MapDR1b                                              |
      IF  a) StoreExplicitly is a good candidate
          b) Number of refs * recompute cost is less costly than
          number of explicit insertions
      THEN   UnfoldDerivedRelation > StoreExplicitly
| End Selection Rule                                                  |
```

```
| SelectionRule  MapDR2a                                              |
      IF  a) MaintainDerivedRelation is a good candidate
          b) UnfoldDerivedRelation is a good candidate
          c) Number of references * recompute cost is high
      THEN   MaintainDerivedRelation > UnfoldDerivedRelation
      References: 5.1
| End Selection Rule                                                  |
```

```
| SelectionRule  MapDR2b                                              |
      IF  a) MaintainDerivedRelation is a good candidate
          b) UnfoldDerivedRelation is a good candidate
          c) Number of references * recompute cost is low
      THEN   UnfoldDerivedRelation > MaintainDerivedRelation
      References: 5.5, 5.8
| End Selection Rule                                                  |
```

```
| SelectionRule  MapDemon1                                             |
       IF  a) MapByConsolidation is a good candidate
       THEN   MapByConsolidation > (CasifyDemon, UnfoldDemon)
       References: 4.3
| End Selection Rule                                                   |
```

```
| SelectionRule  MapConstraint1                                        |
       IF  a) CaisfyConstraint is a good candidate
       THEN   CaisfyConstraint > UnfoldConstraint
       References: 4.7, 4.9, 4.10, 4.12, 4.13, 4.15, 4.16
| End Selection Rule                                                   |
```

```
| SelectionRule  MapConstraint2                                        |
       IF  a) Goal is Map R|require
           b) M1|method is a good candidate
           c) M2|method is a good candidate
           d) M1 eliminates R
           e) M2 does not eliminate R
       THEN   M1 > M2
       [Don't muck around with R if it can be directly eliminated.]
       References: 4.9, 4.12, 4.16
| End Selection Rule                                                   |
```

```
| SelectionRule  MapConstraint3                                        |
       IF  a) Goal is Map R|require
           b) M1|method is a good candidate
           c) M2|method is a good candidate
           d) M1 moves R closer to a non-deterministic choice point
           e) M2 does not eliminate or move R
       THEN   M1 > M2
       [Moving a requirement towards a.nd choice point is good.]
       References: 4.15
| End Selection Rule                                                   |
```

```
| SelectionRule  Map1                                        |
      IF  a) Goal is Map X
          b) M1|method is a non-negative candidate
          c) M1 casifies X
          d) ~∃ a good candidate
      THEN   Select M1
      [If nothing looks very good, try casifying.]
| End Selection Rule                                         |
```

G.12. Purify

PurifyDemon (5.10, 5.14)

```
| SelectionRule  *PurifyDemon                                |
      IF  a) PurifyDemon is a candidate
      THEN   +2
      References: 5.10, 5.14
| End Selection Rule                                         |
```

G.13. Reformulate

ReformulateLocalAsFirst (1.5)

ReformulateLocalAsLast (1.5)

ReformulateEverMoreAsDuring (unused)

ReformulateAsCondByEmbedding (unused)

RenameVar (2.12, 6.7, 6.14)

```
| SelectionRule  *RenameVar                                          |
      IF  a) RenameVar is a candidate
      THEN   +2
      References: 2.12, 6.7, 6.14
| End Selection Rule                                                  |
```

ReformulateActionCall (TextPreprocessor)

ReformulateDerivedObject (1.13)

```
| SelectionRule  *ReformulateDerivedObject                           |
      IF  a) ReformulateDerivedObject is a candidate
          b) Definition of DO reformulatable as P
      THEN   +2
      [If the body of the derived relation looks like it can be made to match the reformulation
      pattern then give method a try.]
      References: 1.13
| End Selection Rule                                                  |
```

ReformulateDerivedRelation (6.9)

```
| SelectionRule  *ReformulateDerivedRelation                         |
      IF  a) ReformulateDerivedRelation is a candidate
      THEN   +2
      References: 6.9
| End Selection Rule                                                  |
```

ReformulateRelativeRetrievalAsLast (1.14)

```
| SelectionRule  *ReformulateRelativeRetrievalAsLast                 |
      IF  a) ReformulateRelativeRetrievalAsLast is candidate
          b) wrt sequence of RS is constructed by appending
      THEN   +2
      References: 1.14
| End Selection Rule                                                  |
```

ReformulateRelativeRetrievalAsFirst (1.14)

```
| SelectionRule  *ReformulateRelativeRetrievalAsFirst                 |
        IF  a) ReformulateRelativeRetrievalAsFirst is candidate
            b) wrt sequence of RS is constructed by prepending
        THEN   +2
| End Selection Rule                                                  |
```

ReformulateAsObject (1.16, 1.20)

SpecializeRandom (4.6)

```
| SelectionRule  *SpecializeRandom                                    |
        IF  a) SpecializeRandom is a candidate
        THEN   +5
        References: 4.6
| End Selection Rule                                                  |
```

ReformulateExistentialTrigger (6.11)

```
| SelectionRule  *ReformulateExistentialTrigger                       |
        IF  a) ReformulateExistentialTrigger is a candidate
        THEN   +2
        References: 6.11
| End Selection Rule                                                  |
```

Method Ordering Rules

```
| SelectionRule  ReformLoc1                                           |
        IF  a) ReformulateLocalAsFirst is a candidate
            b) R|derived-relation is ordered historically by start E|event
        THEN   ReformulateLocalAsFirst > ReformulateLocalAsLast
| End Selection Rule                                                  |
```

| **SelectionRule** ReformLoc2
IF a) ReformulateLocalAsLast is a candidate
b) R|*derived-relation* is ordered temporally by start E|*event*
THEN ReformulateLocalAsLast > ReformulateLocalAsFirst
References: 1.5
| **End Selection Rule**

| **SelectionRule** ReformLoc3
IF a) ReformulateLocalAsFirst is a candidate
b) R|*base-relation* is maintained by simple prepending
THEN ReformulateLocalAsFirst > ReformulateLocalAsLast
| **End Selection Rule**

| **SelectionRule** ReformLoc4
IF a) ReformulateLocalAsLast is a candidate
b) R|*base-relation* is maintained by simple appending
THEN ReformulateLocalAsLast > ReformulateLocalAsFirst
| **End Selection Rule**

G.14. Remove

RemoveFromDemon (5.11, 5.15)

| **SelectionRule** *RemoveFromDemon
IF a) RemoveFromDemon is a candidate
THEN +2
References: 5.11, 5.15
| **End Selection Rule**

RemoveRelation (1.1, 2.1, 3.1)

```
| SelectionRule  *RemoveRelation1                                   |
      IF  a) RemoveRelation is being considered
          b) R's argument is a sequence S
          c) Only one element of S is referenced
      THEN   +2
      [May be able to replace sequence with single object.]
      References: 1.1
| End Selection Rule                                                |
```

```
| SelectionRule  *RemoveRelation2                                   |
      IF  a) RemoveRelation is being considered
          b) R is acting as a temporary variable
      THEN   +2
      [Can get rid of temporary variables]
      References: 2.1
| End Selection Rule                                                |
```

```
| SelectionRule  *RemoveRelation3                                   |
      IF  a) RemoveRelation is being considered
          b) Only use of R is in attribute expressions
      THEN   +2
      [Can replace R with various attributes.]
      References: 3.1
| End Selection Rule                                                |
```

ReplaceRefWithValue (1.12, 1.19, 2.2, 3.2)

```
| SelectionRule  *ReplaceRefWithValue1                              |
      IF  a) ReplaceRefWithValue is being considered
          b) Can find a change to the relatin before its use
      THEN   +2
      References: 2.2
| End Selection Rule                                                |
```

```
| SelectionRule  *ReplaceRefWithValue2                         |
       IF  a) ReplaceRefWithValue is being considered
           b) RR's argument is a sequence
       THEN   -2
       [Unlikely that the entire sequence can be unfolded.]
       References: 1.12
| End Selection Rule                                           |
```

MegaMove (1.2, 1.12, 1.19, 2.2, 3.2)

```
| SelectionRule  *MegaMove1                                    |
       IF  a) MegaMove is being considered
           b) ~∃ derived relation with defintion Y
       THEN   +2
       References: 1.2, 1.12, 1.19, 2.2, 3.2
| End Selection Rule                                           |
```

```
| SelectionRule  *MegaMove2                                    |
       IF  a) MegaMove is being considered
           b) ∃ derived relation with defintion Y
       THEN   -2
       References: 1.12
| End Selection Rule                                           |
```

PostionalMegaMove (1.2, 1.12, 1.19, 2.2, 3.2)

```
| SelectionRule  *PositionalMegaMove                           |
       IF  a) PositionalMegaMove is being considered
       THEN   +1
       References: 1.2, 1.12, 1.19, 2.2, 3.2
| End Selection Rule                                           |
```

RemoveVariable (TextPreprocessor)

RemoveByObjectizingContext (1.2, 1.12, 1.19, 2.2, 3.2)

| **SelectionRule** *RemoveByObjectizingContext |
IF a) RemoveByObjectizingContext is a candidate
b) Y|*positional-retrieval*
THEN +2
References: 1.18
| **End Selection Rule** |

RemoveUnusedAction (1.21, 3.5, 5.11, 5.15)

| **SelectionRule** *RemoveUnusedAction1 |
IF a) RemoveUnusedAction is a candidate
b) A|*update*
c) Supergoal is *Remove* updated relation
THEN good candidate
[To remove a realtion you generally have to show update is unused.]
References: 1.21, 3.5
| **End Selection Rule** |

| **SelectionRule** *RemoveUnusedAction2 |
IF a) RemoveUnusedAction is a candidate
b) Supergoal is *Purify*
THEN +2
[In many cases, unfolded code can be simplified away.]
References: 5.11, 5.15
| **End Selection Rule** |

ReplaceVariableWithValue (TextPreprocessor)

BabyWithBathWater (1.2, 1.12, 1.19, 1.21, 2.2, 3.2, 3.5, 5.11, 5.15)

```
| SelectionRule  *BabyWithBathWater1                              |
        IF  a) BabyWithBathWater is being considered
            b) Y|conditional
        THEN   +0
        References: 1.2, 1.19, 2.2, 3.2
| End Selection Rule                                              |
```

```
| SelectionRule  *BabyWithBathWater2                              |
        IF  a) BabyWithBathWater is being considered
            b) Y|demon
            c) Y in implementable portion
        THEN   -1
        References: 1.2, 1.12, 1.19, 1.21, 2.2, 3.2, 3.5
| End Selection Rule                                              |
```

```
| SelectionRule  *BabyWithBathWater3                              |
        IF  a) BabyWithBathWater is being considered
            b) Y|~{conditional,demon}
        THEN   -2
        References: 1.2, 1.12, 1.19, 1.21, 3.5, 5.11, 5.15
| End Selection Rule                                              |
```

Method Ordering Rules

```
| SelectionRule  RemoveRef1                                       |
        IF  a) MegaMove good candidate
        THEN   MegaMove > PositionalMegaMove
        References: 1.2, 1.19, 3.2
| End Selection Rule                                              |
```

```
| SelectionRule  RemoveRef2                                          |
      IF  a) M1|MegaMove is candidate
          b) M2|MegaMove is good candidate
          c) component-of[Y of M2, Y of M1]
      THEN   M1 > M2
      [Usually better to take as much context with you as possible.]
      References: 1.2, 1.12, 1.19
| End Selection Rule                                                 |
```

```
| SelectionRule  RemoveRef3                                          |
      IF  a) M1|PositionalMegaMove is candidate
          b) M2|PositionalMegaMove is candidate
          c) component-of[Y of M2, Y of M1]
      THEN   M1 > M2
      [Usually better to take as much context with you as possible.]
      References: 1.2, 1.12, 1.19
| End Selection Rule                                                 |
```

```
| SelectionRule  RemoveRef4                                          |
      IF  a) RemoveByObjectizingContext is a good candidate
      THEN   RemoveByObjectizingContext > (MegaMove, PositionalMegaMove)
      References: 1.19
| End Selection Rule                                                 |
```

```
| SelectionRule  RemoveRef5                                          |
      IF  a) BabyWithBathWater is a good candidate
      THEN   BabyWithBathWater > (MegaMove, PositionalMegaMove)
| End Selection Rule                                                 |
```

```
| SelectionRule  RemoveRef6                                          |
      IF   a) ReplaceRefWithValue is a good candidate
      THEN    ReplaceRefWithValue > (MegaMove, PositionalMegaMove)
      References: 2.2
| End Selection Rule                                                 |
```

```
| SelectionRule  RemAct1                                             |
      IF   a) RemoveUnusedAction is a good candidate
      THEN    RemoveUnusedAction > RemoveFromDemon
      [It's worth a try.]
      References: 5.11, 5.15
| End Selection Rule                                                 |
```

G.15. Show

ShowNoChange (4.16)

ConjunctImpliesConjunctArm (4.2)

```
| SelectionRule  *ConjunctImpliesConjunctArm1                        |
      IF   a) ConjunctImpliesConjunctArm is a candidate
           b) Supergoal is Map C|prohibitive-constraint
           c) The conjunct arm A is a good predictor
      THEN    +2
      References: 4.2
| End Selection Rule                                                 |
```

| SelectionRule *ConjunctImpliesConjunctArm2 |

IF a) ConjunctImpliesConjunctArm is a candidate

b) Supergoal is *Map* C|***prohibitive-constraint***

c) The conjunct arm A is a bad predictor

THEN -2

[e.g. A is bad if it acts as idiot light: tells you when something is wrong, but no way to backtrack and make it right.]

References: 4.2

| End Selection Rule |

ShowDysteleological (1.22, 2.14, 3.6)

| SelectionRule *ShowDysteleological |

IF a) ShowDysteleological is a candidate

THEN +2

References: 1.22, 2.14, 3.6

| End Selection Rule |

ShowUpdateGivesValue (2.3)

| SelectionRule *ShowUpdateGivesValue |

IF a) ShowUpdateGivesValue is a candidate

THEN +2

References: 2.3

| End Selection Rule |

ShowNewValueStillValid (2.4)

| SelectionRule *ShowNewValueStillValid |

IF a) ShowNewValueStillValid is a candidate

THEN +2

References: 2.4

| End Selection Rule |

MoveInterveningUpdate (2.5)

```
| SelectionRule   *MoveInterveningUpdate                        |
        IF  a) MoveInterveningUpdate is a candidate
        THEN    +2
        References: 2.5
| End Selection Rule                                            |
```

Method Ordering Rules

```
| SelectionRule   ShowVal1                                      |
        IF  a) M1 | *ShowUpdateGivesValue
            b) M2 | *ShowUpdateGivesValue
            c) M1 computationally closer to R than M2
        THEN    M1 > M2
| End Selection Rule                                            |
```

G.16. Simplify

No rules.

G.17. Swap

SwapStatements (2.9)

```
| SelectionRule   *SwapStatements                               |
        IF  a) SwapStatements is a candidate
        THEN    +5
        References: 2.9
| End Selection Rule                                            |
```

G.18. Unfold

ScatterComputationOfDerivedRelation (3.19, 4.18, 5.6, 5.9, 6.10, 6.19)

```
| SelectionRule  *ScatterComputationOfDerivedRelation                  |
      IF  a) ScatterComputationOfDerivedRelation is a candidate
      THEN   +5
      References: 3.19, 4.18, 5.6, 5.9, 6.10, 6.19
| End Selection Rule                                                   |
```

ScatterComputationOfDemon (6.4, 6.20)

```
| SelectionRule  *ScatterComputationOfDemon                            |
      IF  a) ScatterComputationOfDemon is a candidate
      THEN   +5
      References: 6.4, 6.20
| End Selection Rule                                                   |
```

UnfoldAtomic (2.7, 5.13, 5.16)

```
| SelectionRule  *UnfoldAtomic                                         |
      IF  a) UnfoldAtomic is a candidate
      THEN   +5
      References: 2.7, 5.13, 5.16
| End Selection Rule                                                   |
```

UnfoldSimpleSB (TextPreprocessor)

G.19. Problem Solving Resource Rules

```
| SelectionRule  ReformUnnecessary                                   |
      IF  a) M|method is candidate
          b) M contains a reformulate action A
          c) A is achieved trivially
      THEN   +1
      References: 1.11, 1.14, 1.16, 1.19, 1.20, 4.8, 4.9, 4.11, 4.14,
                     4.15, 5.2
| End Selection Rule                                                 |
```

```
| SelectionRule  RequireReformUnnecessary                            |
      IF  a) Goal is {Map, Casify} R|require
          b) M|method is candidate
          c) M contains a reformulate action A
          d) A is achieved trivially
      THEN   +1
      [Give a bonus to methods which don't need to reformulate a require statement.]
      References: 4.8, 4.9, 4.11, 4.14, 4.15
| End Selection Rule                                                 |
```

```
| SelectionRule  EquivUnnecessary                                    |
      IF  a) M|method is candidate
          b) M contains an equivalence action A
          c) A is achieved trivially
      THEN   +1
| End Selection Rule                                                 |
```

```
| SelectionRule  ReadyToGo                                           |
      IF  a) M|method is candidate
          b) forall actions A of M either 1) A is an Apply,
                  or 2) A is achieved trivially
      THEN   +1
      [If only apply goals left then cheap choice]
      References: 1.11, 1.16, 1.17, 1.22, 2.5, 4.8, 4.9, 4.11, 4.14, 6.5
| End Selection Rule                                                 |
```

```
| SelectionRule  *ShowUnnecessary                              |
       IF  a) M|method is candidate
           b) M contains a Show action A
           c) A is achieved trivially
       THEN   +1
| End Selection Rule                                           |
```

G.20. General Rules

```
| SelectionRule  BurnedOutHulk                                 |
       IF  a) Goal is Remove X from spec
           b) X is a defined strucutre
           c) Method M removes the need for X
       THEN   +2
       References: 1.1, 2.1, 3.1
| End Selection Rule                                           |
```

```
| SelectionRule  FillIn                                        |
       IF  a) Goal is Remove RR|relation-reference from spec
       THEN   Try filling in values within RR's context
       References: 1.2, 1.12, 1.19, 2.2, 3.2
| End Selection Rule                                           |
```

```
| SelectionRule  MapSubOfRemove1                               |
       IF  a) Goal/Supergoal G is Map X
           b) Supergoal of G is Remove X from spec
       THEN   +1
       [A method which keeps X localized facilitates the higher level of goal of removing X.]
       References: 1.10, 1.11
| End Selection Rule                                           |
```

```
| SelectionRule  MapSubOfRemove2                                                  |
        IF  a) Goal/Supergoal G is Map X
            b) Supergoal of G is Remove X from spec
        THEN  -2
        [A method which spreads X out when trying to remove it is counterproductive.]
        References: 1.11
| End Selection Rule                                                              |
```

```
| SelectionRule  DemonsAreGood                                                    |
        IF  a) Goal/Supergoal is Map X
            b) Method M changes X to a demon
        THEN  +1
        [Demons are generally easy to work with.]
        References: 1.11, 4.1, 5.2
| End Selection Rule                                                              |
```

```
| SelectionRule  SubComponent                                                     |
        IF  a) Goal is Reformulate X as P
            b) pattern-match[Y, P, X]
            c) Method M extracts Y from X
        THEN  +2
| End Selection Rule                                                              |
```

```
| SelectionRule  ReformAsExtreme                                                  |
        IF  a) Goal is Reformulate R|relative-retrieval as X=P|positional-retrieval
            b) Method M reforms R as extreme
        THEN  +1
        References: 1.14
| End Selection Rule                                                              |
```

| **SelectionRule** UseConjunctArm |

IF a) Goal is *Show* X|*conjunction* implies Y|*unbound*

b) Supergoal is *Map* C|*prohibitive-constraint*

c) Method M binds Y to arm of X

THEN +2

References: 4.2

| **End Selection Rule** |

| **SelectionRule** CasifyComplexConstruct |

IF a) Goal is *Map* X

b) X is complex

c) Method M splits X into simpler cases

THEN +2

References: 4.4, 4.7, 4.9, 4.10, 4.12, 4.13, 4.15, 4.16, 6.1

| **End Selection Rule** |

| **SelectionRule** CheapRemove |

IF a) Goal is *Remove*

b) M|*method* is candidate

c) forall actions A of M either 1) A is an *Apply*,
or 2) A is achieved trivially

THEN +2

[If you can get rid of something cheaply, do it.]

| **End Selection Rule** |

END

FILMED

5-84

DTIC